A Complete Semantics of $\mathbb{K}$

ANONYMOUS AUTHOR(S)

$\mathbb{K}$ [Roşu and Şerbănuţă 2010] is a framework that allows users to define language specifications, and run programs of these specifications to see traces and behaviors. It also includes some verification tools based on Matching Logic [Roşu and Ştefănescu 2011] to provide methods to analyze the syntax and semantics of a language. $\mathbb{K}$'s mechanism combines ideas from Modular Structural Operational Semantics [Mosses 2004] and Context Evaluation [Felleisen and Friedman 1986] to allow programmers to easily and quickly define programming constructs and investigate the relations and behaviors between the programming languages and different external program environments. Unfortunately, there is neither a explorable $\mathbb{K}$ specification in any credible proof engines nor a paper-based full $\mathbb{K}$ specification that users can easily access.

In this paper we define a reference semantics for $\mathbb{K}$ in the interactive theorem prover Isabelle/HOL[Paulson 1990] developed through the discussion of the $\mathbb{K}$ team to meet what they desire as the semantics of $\mathbb{K}$. The $\mathbb{K}$ specification defines the full behavior of $\mathbb{K}$ and suggests several undesirable behaviors in the current $\mathbb{K}$ implementations ($\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0). To the best of our knowledge, our $\mathbb{K}$ specification covers the most cases in defining the behavior of $\mathbb{K}$ of any existing $\mathbb{K}$ specification. 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 LLVM semantics in $\mathbb{K}$, Java semantics in $\mathbb{K}$ and 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.

ACM Reference format:
Anonymous Author(s). 2017. A Complete Semantics of $\mathbb{K}$. 1, 1, Article 1 (July 2017), 29 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 related to $\mathbb{K}$ and its theories. A lot of popular programming language semantics have been defined and explored largely or fully in $\mathbb{K}$, such as Java semantics [Bogdănaş and Roşu 2015], Javascript semantics, PHP semantics [Filaretti and Maffeis 2014], C semantics [Ellison and Rosu 2012; Hathhorn et al. 2015], LLVM semantics and Python semantics. The user experiences of those scholars defining those languages seemed to agree that they can define programming languages easier then defining those languages in other frameworks such as Isabelle or Coq[Corbineau 2008].

Despite the success of $\mathbb{K}$, there are two 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 ther have been a number of paper published concerning theory related to $\mathbb{K}$, there is no source sufficiently complete to allow for rigorous proofs of properties of the languages defined in $\mathbb{K}$.

XXXX-XXXX/2017/7-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Matching Logic[Roşu and Ştefănescu 2011] and Reachability Logic[Roşu et al. 2013] are theories closely related to $\mathbb{K}$, but have been developed on top of the tools provided by $\mathbb{K}$. Moreover, the fact that early versions of $\mathbb{K}$ had features that were dropped in intermediate versions, only yo 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 addresses these concerns and can form the foundation of tools for the maintenance, revision and expansion of $\mathbb{K}$. Moreover, 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.

Even though $\mathbb{K}$ has been used in the definition 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, such as the LTL tools associated with the C semantics in $\mathbb{K}$, of researchers extending the language specifications in $\mathbb{K}$ to create various tools for analyzing program properties. However, as of now, only an experimental Reachability Logic 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 defined in $\mathbb{K}$, it provides no support for formal reasoning about the languages it defines. Having a formal $\mathbb{K}$ specification in Isabelle is a first step towards providing $\mathbb{K}$ with additional power of full formal proofs and marrying a system with considerable experience in language specification with a system with an impressive history proving results of mathematics and complex computer systems.

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 and generated parsers for the language specified, compiler and runtime interpreter, tools for generating test cases, and more. In our specification, we focus on the meaning of $\mathbb{K}$ as a system for defining a programming language by specifying the syntax and execution behavior of programs of that language.

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 semantics 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.

Since $\mathbb{K}$ provides compilation and interpretation of the compiled code for a $\mathbb{K}$ language specification, to specify $\mathbb{K}$, in addition to the generic $\mathbb{K}$ abstract syntax, we need a second syntax for the target of the compiled code, $\mathbb{K}$ IR. Starting from a generic $\mathbb{K}$ abstract syntax, we define compilation and sort checking for $\mathbb{K}$ specifications. Compilation transfers a abstract syntax to $\mathbb{K}$ IR syntax, while eliminating some meaningless $\mathbb{K}$ definitions that are trivial enough to be captured while translating languages. Sort checking step is applied to the $\mathbb{K}$ IR syntax, and rules out further meaningless language definitions. Sort checking is actually used in two different ways in $\mathbb{K}$. It is applied statically on the language definition rewriting rules and dynamically on given programs and their environments in every step of computation. All terms that in $\mathbb{K}$ are divided into two different classes: function terms and normal terms. The final destinations of function rules should always output normal terms or $\mathbb{K}$ built-in terms, but this can include syntactic objects, and such

as poses extra complications for the semantics. All the semantic rules definable in $\mathbb{K}$ fall into four different categories: macro rules, function rules, anywhere rules and normal rules. Our semantics has to have special cases for each of these, for each of compilation, sort checking and execution.

Given that there is no other complete documentation of the semantics of $\mathbb{K}$, what measures did we use to assure that ours is correct? The first angle of attack on this is to be fully aware of the literature pertaining the theory and applications of $\mathbb{K}$. The second angle is to talk to the developers and users to learn their intents and experiences. The last approach is to export code from the theorem prover from the specification to generate an interpreter for $\mathbb{K}$, and test the exported code against already pre-existing language specifications and their programs. During our development we found over 20 issues that by the methods above can be classified as bugs. In Section 7, we will review the details of some of these issues and the evidence that are incorrect / undesirable. Applying the last approach to several large language definitions and running their torture tests gives strong evidence that our semantics does indeed capture the intended behavior of K.

2 THE STRUCTURE OF A $\mathbb{K}$ SPECIFICATION

The main purpose of the $\mathbb{K}$ framework is to provide users with an intuitive, concise, and testable way to define language specifications. The functionality of $\mathbb{K}$ is described generally in "An Overview of the K Semantic Framework" [Roşu and Şerbănuţă 2010]. In $\mathbb{K}$, the semantic objects that are used to describe a program's execution are viewed as living in a collection of `cells`, with the union of all the user-defined cells being called `bag`. The idea of the cells is that they represent various resources with which the program interacts, such as the stack, the program counter, the registers, the global memory, IO buffers, etc.

Among these `cells`, there is a special cell, named `kCell`, having list type, and consisting initially of a program. In general, `kCell` consists of a list of suspended partial computations headed by the computation to be processed next. The cell `kCell` acts like a stack machine, typically only operating on the computation on the head of the list, or the top two elements on completion of the top computation.

$\mathbb{K}$ allows users to define a sort, named `KResult`, to tell what are the values of an evaluation. The sort `KResult` plays a critical role in the meaning of `heat/cool` rules. The idea of `heat/cool` rules originated with Context Evaluation [Felleisen and Friedman 1986]. In $\mathbb{K}$, `heat` rules pull out from the head of the list a subcomputation to become a new head, provided the subcomputation is not a `KResult`, and leave a *redex hole* in the previous head. In the opposite direction, `cool` rules merge a `KResult` value at the head of the `kCell` list into the *redex hole* in the second element of the `kCell` list. There are other types of $\mathbb{K}$ rules that act on different parts of process environment pieces, including `kCell`, and we will talk about them later in this section. In addition to these normal types of rules to define a language specification, $\mathbb{K}$ provides users other types of rules to define language specifications easily, such as providing function terms and rules for users to reason about sub-pieces of a given program state.

Every version of K [Şerbanuţă and Roşu 2010; Lucanu et al. 2012] contains a user friendly front-end language that is translated through multiple steps to a final small back-end language. Language interpretation is defined on the back-end language. In addition, the pre-existing $\mathbb{K}$ parser deals with a user defined specification in two phases. It first reads all the syntactic definitions from a given specification and generates a list of `kSyntax` entities in the form of $\mathbb{K}$ **SYNTAX**, then $\mathbb{K}$ generates a symbol table for all constructs, including user defined ones. Using the symbol table, the pre-existing parser then parses the semantic rules in the given specification. Based on this knowledge, we model the $\mathbb{K}$ semantics in a similar manner by using a front-end syntactic definition language and a front-end semantic definition language, both in the form of abstract

syntax, named $\mathbb{K}$ **SYNTAX** and $\mathbb{K}$ **CORE** respectively, shown in Figure 1 and 2, and a back-end language, again in the form of abstract syntax, named $\mathbb{K}$ **IR**, shown in Figure 3. Users define a language specification in the concrete syntax of $\mathbb{K}$ that is translated into a form of $\mathbb{K}$ **CORE** by the pre-existing parser of $\mathbb{K}$. Through a compilation process described in Section 3, the specification in $\mathbb{K}$ **CORE** is further translated into a representation in $\mathbb{K}$ **IR**. It is the $\mathbb{K}$ **IR** form that is used to defined the sort system and semantics of $\mathbb{K}$. The complication process eliminates language specifications that use unsupported features in $\mathbb{K}$, which means that the compilation function is a partial function on $\mathbb{K}$ **CORE**.

2.1 $\mathbb{K}$ SYNTAX and $\mathbb{K}$ CORE Data Structures

$\mathbb{K}$ ***SYNTAX***. Users define the syntactic categories for their language specifications in $\mathbb{K}$ as $\mathbb{K}$ sorts by giving a collection of `kSyntax` directives. The language $\mathbb{K}$ **SYNTAX** describes the sorts of the object language specification and their related data constructs, and the subsort relation, all using concrete syntax. The pre-existing parser processes the user input to find the syntax directives, and parses them into a list of `kSyntax` data structures. The data structures produced will contain the grammar productions for the concrete syntax for the object language construct. During compilation, these productions will be processed into a symbol table that will be passed to a second parser for processing the rules of the specification.

There are three layers of languages used in discussing the $\mathbb{K}$ semantics. The bottom-most one is that of the object language being specified in $\mathbb{K}$, with its own syntax and semantics to be expressed by $\mathbb{K}$. The next layer up is $\mathbb{K}$ itself. But above that there is another layer, that of the language(s) we, and $\mathbb{K}$ developers, are using to express the syntax and semantics of $\mathbb{K}$, the meta-meta-languages such as mathematics, Isabelle, OCaml, Java, Maude, etc. It is important to distinguish between the types that are used in the meta-meta-language for representing the constructs used to explain the semantics of $\mathbb{K}$, such as `kLabel`, `kItem`, `kList`, `k`, `aList`, `set`, `bag` and `map` in the $\mathbb{K}$ **CORE** language, and the sorts of $\mathbb{K}$ itself including `KLabel`, `KItem`, `KList`, `K`, `List`, `Set`, `Bag` and `Map`, and user defined sorts. The main role of `kSyntax` is to allow the user to define the subsorts of `kItem`.

In $\mathbb{K}$ data structures, several base datatypes that may appear in any data structure are `sortId` (representing sort identifiers), `metaId` (representing meta variable identifiers), `cellId` (names for cells), `regex` (representing regular expressions that define families of labels being used in a specification), `labelId` (representing user defined constructor names), `symbol` (representing terminals or nonterminals that support syntactic definitions), `synAttrib` (representing attributes giving syntactic or semantic properties for the syntactic definition to which they are attached), `ruleAttrib` (representing attributes giving semantic properties for the rule to which they are attached) and `feature` (representing restrictions on specific `cells`). We will examine more about different kinds of `synAttrib`, `ruleAttrib` and `feature` in Section 2.3.

There are four different types of `kSyntax` directives: **Syntax**, **Token**, **ListSyntax** and **Subsort**. The construct **Syntax**, gives the concrete syntax of object language constructs. The first argument to **Syntax** defines the user defined sort of the construct. The second (`symbol nelist`) is effectively the right-hand side of a production in a grammar for the construct. The third argument (`synAttrib list`) argument is as described above. The construct **Token** allows users to define a potentially infinite set of object language constant constructs by the regular expression argument `regex`, while the sort of the elements in the set is the first argument. User defined subsort information is given using **Subsort**, where its first argument is the subsort and the second argument is the supersort.

The definition **ListSyntax** `sortId sortId string (synAttrib list)` gives users the ability to define a cons list sort where the first argument is the sort name and the second defines the element sort in the list. Through this definition, users can get two constructs in the symbol table:

one as a cons list operator, the other as a unit list operator. The `string` is the separator for the list, and is a part of the `kLabel` term generated for the cons list operator. Different $\mathbb{K}$ versions have different implicit equational rules associated with the user defined list constructs. Before $\mathbb{K}$ version 3.0, the $\mathbb{K}$ user defined list data structure has implicit identity and associativity equational rules. In $\mathbb{K}$ versions 3.0 - 3.6, there is neither identity nor associativity. $\mathbb{K}$ 4.0 has identity and a problematic implementation fo associativity that either counts as a bug or a design problem. Due to these confusions, in this paper, we fix the user defined list constructs to have no identity or associativity since it is easy to model, and they would be easy to add in the future.

```
    sortId = ⟨capitalized name⟩          metaId = ⟨capitalized name⟩          regex = ⟨regular expression⟩
   labelId = string     key = ∗ | ?     symbol = Terminal string | NonTerminal sortId     std = In | Out
synAttrib = Strict (nat nelist) | Seqstrict | Function | Klabel labelId | Bracket | ...
ruleAttrib = Macro | Anywhere | Owise | Transition | ...          feature = Multiplicity key | Stream std | ...
   kSyntax = Syntax sortId (symbol nelist) (synAttrib list)   | Subsort sortId sortId
           | ListSyntax sortId sortId string (synAttrib list) | Token sortId regex
```

Fig. 1. $\mathbb{K}$ **SYNTAX** Data Structure

```
     const = IntConst int | FloatConst real | StringConst string | BoolConst bool | IdConst string
'a rewrite = Term 'a | Rewrite 'a 'a          cellId = ⟨name⟩
    kLabel = KLabelC labelId | IdKLabel metaId | KLabelFun funApp          funApp = FunApp kLabel (kList rewrite)
     kList = KListCon (kList rewrite) (kList rewrite) | UnitKList | KListItem bigKBag | IdKList metaId
  bigKBag = BigK bigK | ABag (bag rewrite) | AKLabel (kLabel rewrite)
     bigK = AK (k rewrite) | AList (aList rewrite) | ASet (set rewrite) | AMap (map rewrite)
    kItem = KItemC (kLabel rewrite) (kList rewrite) sortId | Const const sortId | IdKItem metaId sortId | HOLE sortId
        k = ⇝ (k rewrite) (k rewrite) | UnitK | SingleK (kItem rewrite) | IdK metaId | KFun funApp
    aList = ListCon (aList rewrite) (aList rewrite) | UnitList | ListItem (k rewrite) | IdList metaId | ListFun funApp
      set = SetCon (set rewrite) (set rewrite) | UnitSet | SetItem (k rewrite) | IdSet metaId | SetFun funApp
      map = MapCon (map rewrite) (map rewrite) | UnitMap | MapItem (k rewrite) (k rewrite)
          | IdMap metaId | MapFun funApp
      bag = BagCon (bag rewrite) (bag rewrite) | UnitBag | IdBag metaId | □ cellId (feature list) (bag rewrite)
          | ◯ cellId (feature list) (bag rewrite) | Cell cellId (feature list) bigK
    kRule = KRule (k rewrite) [kItem] (ruleAttrib list)          | KItemRule (kItem rewrite) [kItem] (ruleAttrib list)
          | Context kItem [kItem] (ruleAttrib list)              | KLabelRule (kLabel rewrite) [kItem] (ruleAttrib list)
          | ListRule (aList rewrite) [kItem] (ruleAttrib list)   | MapRule (map rewrite) [kItem] (ruleAttrib list)
          | SetRule (set rewrite) [kItem] (ruleAttrib list)      | BagRule (bag rewrite) [kItem] (ruleAttrib list)
```

Fig. 2. $\mathbb{K}$ **CORE** Data Structure

$\mathbb{K}$ ***CORE Language***. In giving semantic definitions, users need a language to represent the datatypes that provides different program resources used by the specification, a list of rules supplying the semantic definitions describing the execution of object programs, and a prototype giving the initial configuration of the overall environment in which the program executes. These are the design goals of $\mathbb{K}$ **CORE**.

The datatype `kRule` represents the top level syntactic category of $\mathbb{K}$ **CORE** for expressing rewrites for the various data structures in the language specification, and directly utilizes most of the other datatypes of $\mathbb{K}$ **CORE**. There is one `kRule` constructor for each type admitting rewrites to facilitate rejection of ill-formed specifications. There is also a `kRule` construct with constructor **Context** that acts as a prototype to generate `heat/cool` pairs, which is similar to the **Strict** `synAttrib` (described in Section 2.3). In fact, any **Strict** `synAttrib` can be written in a list of **Context** rules. The first argument of a **Context** rule must be a `kItem` term and contain a *redex hole* if there are no rewrites in the term and two *redex holes* if there is a rewrite. The term acts as a pattern for $\mathbb{K}$ to generate `heat/cool` pairs, while the second `kItem` term is a possible rule condition.

Other `kRules` follow a convention of *RuleLabel* (*T* `rewrite`) `[kItem]` (`ruleAttrib list`), where *RuleLabel* is the constructor of each `kRule` construct, *T* is different datatypes for different `kRules`, `kItem` is a possible rule condition and `ruleAttrib` contains some semantic attributes that affect the behaviors of the rule defined by the (`ruleAttrib list`) argument.

Rewrite rules, possibly with side conditions, traditional have a left-hand term with variables that acts as a pattern, and a right-hand term whose variables are a subset of those on the left. If a term

matches the left hand side pattern of a rule and fulfills the condition, then the term can transition to another term that matches the right hand side pattern. However, $\mathbb{K}$ provides a flexible and generic rewrite construct (*T* `rewrite`) that allows for on-the-fly rules, where rewrites can appear as any sub-terms in the first argument as long as they do not contain sub-terms that have rewrites. For example, in constructing a rule involving heaps and threads, users can have an overall term with a rewrite in the cell that represents heaps and another rewrite in the cell that represents threads, with both cells containing the rewriting symbol. In Section 3, we translate these rules in $\mathbb{K}$ **CORE** into conventional rules without rewrites appearing anywhere in terms in $\mathbb{K}$ **IR**.

The datatypes `kLabel`, `kItem`, `kList`, `k`, `aList`, `set`, `bag` and `map` are mutually recursive, and jointly are called terms in this paper. $\mathbb{K}$ **CORE** terms have three major roles: acting as the first argument of a `kRule`, acting as a prototype to define the initial program state of a given specification and acting as a real ground term program state representing an intermediate step of execution.

The $\mathbb{K}$ **CORE** term syntax is a large collection of datatypes that cooperate to accomplish the jobs above and can be understood by the definition in Figure 2. There are several key points that we need to mention about these types. First, the datatype `kItem` represent object language constructs, with `kLabel` giving the constructs abstract syntax constructor name, and `kList` giving a list of arguments. There is a special term **HOLE** `sortId` that represents a *redex hole* similar to the holes in the Context Evaluation framework. The `sortId` at the end of every construct in `kItem` is the upper bound restriction on the term.

The built-in $\mathbb{K}$ **CORE** aggregate datatypes `k`, `aList`, `map`, `set` and `bag` provide users convenience in defining some usual data structures used in some process environment pieces. These datatypes, as well as `kList` are designed as list constructs, though they are associated with different implicit equational rules. We need to take in account the syntactic congruence for these datatypes when we define the compilation and pattern matching algorithm for these datatypes. The types `k`, `aList`, `kList` have both identity and associativity implicit equational rules. The types `set` and (`irSetElem list`) have identity, associativity, idempotence and commutativity, while `map` and (`irMapElem list`) have identity, associativity, idempotence, commutativity and being functional (every key uniquely defines a value). The types `bag` and (`irBagElem list`) have identity, associativity and commutativity.

Unlike in many functional languages, the list constructs use list concatenation (a.k.a. list appending) instead of list consing because $\mathbb{K}$ inherits the idea from Maude that the sort of an element of a list is a subsort of the list, so every element should be treated as a singleton list. Additionally, to support the equational rules of associated with lists, it is more convenient to define them as list concatenation parsed as a tree structure, then compile them to $\mathbb{K}$ **IR** in a cons list structure.

The datatype `bag` represents the interactions of different program resources including the program code of a specification. $\mathbb{K}$ requires a specification to include a `bag`, called a configuration, describing the full initial program environment, including the program, used for evaluation. This configuration supplies the type of the different program resources. With this, $\mathbb{K}$ allows users to mention minimum environment pieces when designing their rules, an idea borrowed from modular structural operational semantics [Mosses 2004]. There are three different types of `cells` data structures: a square cell □ `cellId (feature list) (bag rewrite)`, a circle cell ◯ `cellId (feature list) (bag rewrite)` and a real cell **Cell** `cellId (feature list) bigK`. Circle cells can only be used in rules, not in definitions of configurations or process environment pieces. Circle cells allow users to define rules describing the relations among small portions of bags that are successors of the circle cells, while square cells force users to mention all cells, except that users can use a `bag` metavariable to represent a list of bags, while users are not allowed to use metavariables in a circle cell.

```
      irKLabel = KLabelIR labelId | KLabelFunIR irFunApp | IdKLabelIR metaId
       irFunApp = FunAppIR irKLabel (irKListElem list)
   irKListElem = KListElemIR irBigKBagLabel | KListIdIR metaId
irBigKBagLabel = BigBagIR irBigKBag | KLabelIR irKLabel
     irBigKBag = KIR (irKElem list) | ListIR (irListElem list) | SetIR (irSetElem list) | MapIR (irMapElem list)
               | BagIR (irBagElem list)
        irKElem = KElemIR irKItem | KIdIR metaId | KFunIR irFunApp
        irKItem = KItemIR irKLabel (irKListElem list) (sortId set) |IdKItemIR metaId (sortId set) | HOLEIR (sortId set)
    irListElem = ListElemIR (irKElem list) | ListIdIR metaId | ListFunIR irFunApp
     irSetElem = SetElemIR (irKElem list) | SetIdIR metaId | SetFunIR irFunApp
     irMapElem = MapElemIR (irKElem list) (irKElem list) | MapIdIR metaId | MapFunIR irFunApp
     irBagElem = BagElemIR cellId (feature list) (irBigKBag list) | BagIdIR metaId
      irFunPat = KFPat (irKListElem list) * (irKElem list)       | KLabelFPat (irKListElem list) * irKLabel
               | KItemFPat (irKListElem list) * irKItem          | ListFPat (irKListElem list) * (irListElem list)
               | SetFPat (irKListElem list) * (irSetElem list)   | MapFPat (irKListElem list) * (irMapElem list)
       irKRule = KNormalRule (irKElem list) (irKElem list) irKItem bool
               | BagNormalRule (irBagElem list) (irBagElem list) irKItem bool
               | FunRule labelId ((irFunPat * irKItem) list) [irFunPat * irKItem]
               | MacroRule labelId (irKListElem list) (irKElem list)
               | AnywhereRule labelId (irKListElem list) (irKElem list) irKItem
```

Fig. 3. $\mathbb{K}$ **IR** Data Structure

The datatype `funApp` defines a function term. Function terms are only allowed to have output types `kLabel`, `kItem`, `aList`, `set`, `map` and `bag`. Function terms have similar structure to `kItems`. Both include a `kLabel` and a `kList`. However, the function flag in the symbol table tells whether the `kLabel` of a term is a function label. Finally, `const` allows users to use built-in datatypes such as integers, floats, program identifiers, booleans and strings. It is translated to a term whose `kLabel` containing the information about the `const` and whose `kList` is an empty list in $\mathbb{K}$ **IR**.

2.2 $\mathbb{K}$ IR Data Structures

The language $\mathbb{K}$ **IR** defines exactly the set of terms for which $\mathbb{K}$ gives semantics meanings. It has a similar structure to $\mathbb{K}$ **CORE**, but eliminates all aspects of concrete syntax. $\mathbb{K}$ **IR** has a rule datatype `irKRule`, and the compilation step translates all `kRules` to it. These two types have similar use with different design strategies. Where different `kRule` constructs are defined in terms of datatypes, `irKRules` are divided in terms of functionalities. For example, once a list of function rules are identified as having a function label, they are put into a **FunRule** construct. Once a rule cannot be identified as either a **FunRule**, **MacroRule** or a **AnywhereRule**, it is classified as a normal rule and distributed into either a **KNormalRule** or a **BagNormalRule** based on the type of its first argument. In $\mathbb{K}$ **IR**, only **KNormalRule**s and **BagNormalRule**s are classified as normal rules and only a normal rule is considered as a real computation step in the semantics of $\mathbb{K}$. The `bool` value in the last position of **KNormalRule** and **BagNormalRule** indicates if the rule is a **Transition** rule that causes potential nondeterminism in `K_search`. Another difference between `irKRule` and `kRule` is that rules in `irKRule` always have the format: *RuleLabel L R C*, where *L* is the pattern of the rule, *R* as the expression of the rule, and *C* is the condition of the rule. In particular, none of *L*, *R* or *C* contains any rewrites.

The terms of $\mathbb{K}$ **IR** are largely similar to $\mathbb{K}$ **CORE** terms except three differences. First, the binary tree structure in `kList`, `aList`, `set`, `map` and `bag` are compiled a list of `irKListElem`, `irListElem`, `irSetElem`, `irMapElem` and `irBagElem`. Second, the different cells in $\mathbb{K}$ **CORE** are unified into the one type of `irBagElem` lists. Third, the sort restrictions in `irKItem` terms are a set of sorts in $\mathbb{K}$ **IR** representing maximal lower bound sorts on the terms. The datatypes (`irKElem list`), (`irKListElem list`), (`irListElem list`), (`irMapElem list`), (`irSetElem list`) and (`irBagElem list`) are the images of the datatypes `k`, `kList`, `aList`, `map`, `set` and `bag` in $\mathbb{K}$ **CORE**, and they have corresponding equational rules explicitly for them during compilation.

The set of allowed syntactic constructs for an expression, a pattern and a program is different in $\mathbb{K}$ **IR** than in $\mathbb{K}$ **CORE**. Programs are ground term $\mathbb{K}$ **IR** constructs. Patterns have some restrictions. First, if a function term appears in a pattern, it can only appear at the top most level so that the term can be recognized as a part of a function rule. Second, for any list construct described above, $\mathbb{K}$ allows only one variable at the list or list element level. The types (`irKElem list`), (`irKListElem list`) and (`irListElem list`) have no commutativity, so the variable can happen anywhere in a list, while we can restrict any variable in a (`irSetElem list`), (`irMapElem list`) or (`irBagElem list`) list to the end of the list, since these types support commutativity. These restrictions are imposed during compilation to $\mathbb{K}$ **IR**.

2.3 Attributes and Features

The attributes `synAttrib` and `ruleAttrib` define useful observable behaviors for $\mathbb{K}$ constructs and rules, while `feature` defines these behaviors for a cell. We omit many of the attributes and features in this paper since they are used only by some of the $\mathbb{K}$ tools. For example, `synAttrib` contains precedence and associativity for parsing and `feature` contains a construct for defining pretty printing colors for a $\mathbb{K}$ specification. We only list attributes and features related to defining the semantics of $\mathbb{K}$.

A `synAttrib` **Strict** (`nat nelist`) means that $\mathbb{K}$ generates a pair of `heat\cool` rules for each element of the syntactic construct indicated by the list of non-terminal positions associated with the **Strict** attribute. A **Seqstrict** attribute is similar to a **Strict** attribute but the generated `heat\cool` rules are evaluated in the order of the non-terminal positions of the syntactic construct the **Seqstrict** attribute is pointing at. A **Function** attribute basically tells $\mathbb{K}$ that the syntactic construct is a function and that the syntactic construct and all the rules that contain a top most `kLabel` defined by the syntactic construct should be considered as function rules. A **Klabel** L attribute changes the `kLabel` name of a given construct to L. In $\mathbb{K}$, the `kLabel` name uniquely defines a construct in rules and programs. If users need to define overlapping constructs, they need to change these overlapping construct `kLabel` names to be all different. A **Bracket** attribute allows users to define a construct to be a bracket construct such as parentheses in C or Java. A `ruleAttrib` **Macro** defines a rule to be a **MacroRule**, while a **Anywhere** `ruleAttrib` defines a rule to be a **AnywhereRule**. A **Owise** `ruleAttrib` is useful only in function rules. For a given function construct, there is only one **Owise** in all its rules and it defines the behavior if all rules fail to match against a given program piece. A **Transition** attribute can be placed in a rule to indicate that the rule is one of the non-deterministic rules used by $\mathbb{K}$`_search` when generating traces. If non-deterministic rules are not labeled as **Transition**, $\mathbb{K}$`_search` only chooses one of the rules to execute when it faces a non-deterministic choice.

Two useful features are **Multiplicity** and **Stream**. The former one can have $*$ or ? types. The $*$ type **Multiplicity** is used to define rules to generate multiple `cells` of the same name, which is useful in defining threads. The ? type **Multiplicity** is used to define rules to generate zero or one `cell` of the same name. **Stream** is used to define a `cell` for an input or an output channel and $\mathbb{K}$ will simulate the behaviors of IO channels when running programs for a language definition.

3 COMPILATION FROM $\mathbb{K}$ CORE TO $\mathbb{K}$ IR

$\mathbb{K}$ is a programming language that serves both the purpose of defining language specifications and running programs of the defined language to test the language specifications. Hence, the front-end $\mathbb{K}$ **CORE** language is designed to support these two different tasks. It is necessary to translate the front-end $\mathbb{K}$ **CORE** into a back-end $\mathbb{K}$ **IR** language in order to remove some of the front-end features and distinguish the language specifications and from programs for these specifications. In

addition, by compiling to $\mathbb{K}$ **IR**, we can eliminate some ill-formed language definitions and have a very simple sort system, pattern matching algorithm and evaluation rules.

The procedure of compiling a language definition rewritten in $\mathbb{K}$ **SYNTAX** and $\mathbb{K}$ **CORE** format to a form in $\mathbb{K}$ **IR** can be viewed as a partial function *trans* that translates a language specification or a program in $\mathbb{K}$ **SYNTAX** and $\mathbb{K}$ **CORE** into a form in $\mathbb{K}$ **IR**. It contains several sub-steps. The first one is a function $trans^{syn}$ that generates a symbol table, a set of `heat\cool` rules that are defined by the **Strict** and **Seqstrict** syntactic attributes for some constructs and, a subsort graph from **Subsort** constructs. The next step is to have a partial function $trans^{r}$ to translate a set of `kRules` in $\mathbb{K}$ **CORE** into a set of `irKRules` in $\mathbb{K}$ **IR**, and to translate the `bag` type configuration for a language specification to a (`irBagElem list`) type entity. Another partial function $trans^{p}$ translates a program together with the configuration information into a program state by filling the variables in the configuration with the input program and other input arguments. Finally, $trans(spec) = (trans^{r}(trans^{syn}(spec)), trans^{p}(trans^{syn}(spec)))$. All these different sup-steps involve a lot of checks to eliminate some ill-formed language definitions or programs. The compilation process is not the final step before an evaluation can be defined on $\mathbb{K}$ specifications. Two other processes related to sort checking need to be applied after the compilation step and **MacroRules** also affect rules in a language specification.

3.1 Translating Syntactic Definitions

The three tasks of $trans^{syn}$ require only the list of `kSyntax` in a language specification. The symbol table is generated from the **Syntax**, **Token** and **ListSyntax** constructs in the list. The symbol table has type (`sortId` $*$ (`sortId list`) $*$ (`labelId` | (`labelId` $\Rightarrow$ `bool`)) $*$ *bool*), where the first element is the target sort of a given construct, the second argument gives the sorts of the construct arguments. Depending on whether the syntactic definition is defined by a **Token** `sortId regex`, a given syntactic definition can have a `kLabel` constructor name or a potentially infinite set of `kLabel` names (represented by a function) in the third position of an entry of the symbol table. The boolean value is a flag indicating if the label defined by a syntactic definition is a function label.

To build a sntax table entry from a syntactic definition with the key word **Syntax**, we take the target sort for the first entry, and get the construct argument sort list from the non-terminals in the argument list of the syntactic definition. Then we generate a `kLabel` name by using the `genKLabel` function, which takes a list of symbols and converts them into a `kLabel` name, and lastly, we see if there is a **Function** attribute in the (`synAttrib list`). If a **Klabel** L attribute exists in the (`synAttrib list`), $\mathbb{K}$ uses the `kLabel` name L as the `kLabel` name of the entry instead of using the `genKLabel` function to generate one. Translating a syntactic definition with the key word **Token** is the same as with **Syntax** except that we need to generate a boolean function to tell if a `kLabel` name belongs to this entry by using the `regexPar` function, which is a regular expression matcher, together with the regular expression defined in the **Token** construct. Translating a syntactic definition with the key word **ListSyntax** generates two entries. It generates a user defined unit list entry and a user defined cons list entry. Their target sorts are the same, while the former entry has no argument but the latter one has two arguments. If there is a **Strict** attribute in the (`synAttrib list`) of the **ListSyntax** syntactic definition, the **Strict** attribute is assumed to have meanings for the cons list entry. If users define a **Klabel** L attribute in the (`synAttrib list`), the `kLabel` name L is the one for the cons list entry.

The hard parts of the translation are applying checks to eliminate some ill-formed $\mathbb{K}$ **CORE** definitions and programs, generating `heat\cool` rule pairs and a subsort graph. In the following paragraphs, we talk about several checks, the combination of these checks, and the processes to

generate heat\cool rules and the subsort graph, which together form the functionality of the function $trans^{syn}$.

Function, Strict and Seqstrict Checks. First, **Function**, **Strict** and **Seqstrict** cannot be in the same (synAttrib list) of a syntactic definition. In addition, **Strict** requires a natural number following it, which provides a user friendly way to write a pair of heat\cool rules for the argument position specified by the natural number in the argument list, so the number must be in the range from one to the length of the argument list.

Sort Requirement. All target sorts of syntactic definitions must be either user defined target sorts or in the set of {KItem, KResult}, if the syntactic definition does not introduce a function term, or in the set of {KItem, KResult, K, KLabel, List, Set, Map}, if the syntactic definition does introduce a function term. The definitions of all user defined target sorts include the target sorts of the syntactic definitions of the key words **Syntax**, **Token** and **ListSyntax**, and the right hand side sorts in the **Subsort** definitions.

Other Important Checks. In $\mathbb{K}$, we cannot have overlapping kLabel names. The language specification must be rejected if the genKLabel function generates the same kLabel name twice for a language definition. A check must take place in the translation function to implement this mechanism. In addition, $\mathbb{K}$ does not allow users to define two list syntaxes by using the key word **ListSyntax** with the same target sort. One of the consequences is that a user defined list cannot have two different sorts of elements in it. Finally, we need to delete syntactic definitions that have the Bracket attribute in (synAttrib list) since they are removed from the abstract syntax tree by the pre-existing parser and have no semantic meaning. Before that, we need to make sure that any syntactic definitions with Bracket attribute always have one target sort and one argument sort and they are the same. Otherwise, it would introduced an inappropriate subsort relation.

Generating* heat\cool *rule pairs. A pair of heat\cool rules are generated for every position of a syntactic entry's argument list that has the **Strict** or **Seqstrict** attributes. Generating heat\cool rules follows a simple pattern for **Strict** attributes. First, we take a kLabel name and its argument types for an entry in the symbol table and merge them into an irKItem pattern t by applying a **KLabelIR** constructor to the kLabel name fresh variables representing every argument in the argument type list. Second, we take the natural number list of the **Strict** attribute of the syntactic definition. For each natural number from list, we find the corresponding position in the argument list of the kItem pattern. We split the kItem pattern into a redex with a variable of the sort the original positon of the redex and context pair by inserting a **HOLE** T into the corresponding position in the kItem argument list if the position has sort T. We now have a redex and context pair as (**IdKItemIR** X T, t**[HOLE** T**]**), then we generate a heat rule and a cool rule as follow:

- **KNormalRule** [**KElemIR** t, **KIdIR** XL]
  [**KElemIR** (**IdKItemIR** X T), **KElemIR** t**[HOLE** T**]**, **KIdIR** XL]
  (¬isKResult(t)) false
- **KNormalRule** [**KElemIR** (**IdKItemIR** X KResult), **KElemIR** t**[HOLE** T**]**, **KIdIR** XL]
  [**KElemIR** t, **KIdIR** XL] true false

where isKResult is a function checking if a term has supersort KResult, X is a fresh variable for the *redex hole* position, and XL is the tail of the computation sequence in the kCell. The term t**[HOLE** T**]** represents a term t with a *redex hole*. Generating **Seqstrict** only requires small changes in the above heat rule pattern. First, we generate heat\cool rule pairs for all positions for a construct attributed with **Seqstrict**. Second, for generating a pair of heat\cool rules for a

specific position in an argument list, all arguments to the left of the position have sort of `KResult` instead of their own position sorts.

Generating Subsort Graph. If we treat all sorts of a language specification as vertices, the syntactic directives with the key word **Subsort** are basically the directed edges of a graph describing the subsort relation, and a simple scan of the specification suffices to generate a data structure for that graph. However, it is necessary for the subsort graph to include additional implicit $\mathbb{K}$ subsort relations, including that `KItem` is a supersort for `KResult` and all user defined sorts, and that `K` is a supersort for `KItem`. Once created, the graph must be subjected to sanity checks.

First, we check whether there are cycles in the subsort graph by checking if there is a self edge in its transitive closure. Second, `KItem` can only be a subsort of `K`; users cannot define declare `KItem` as a subsort of another sort. Third, the built-in sorts `K`, `Bag`, `KList`, `List`, `Map`, `Set` and `KLabel` cannot have any subsort relation among themselves, or with any other sort, with the exception of `K` being a supersort of `KItem`, `KResult` and user defined sorts. If a language specification violates the above conditions, $\mathbb{K}$ rejects the language specification.

3.2 Translating Terms

Translating terms means translating data structures of type `kLabel`, `kItem`, `kList`, `k`, `aList`, `bag`, `set` and `map` in $\mathbb{K}$ **CORE** into data structures of type `irKLabel`, `irKItem`, (`irKListElem list`), (`irKElem list`), (`irListElem list`), (`irBagElem list`), (`irSetElem list`) and (`irMapElem list`) in $\mathbb{K}$ **IR**. Translating specification rules and programs both depend on translating terms. We refer to the target data structure of patterns as t^{pat} in $\mathbb{K}$ **IR**, the target data structure of expressions as t^{e} and the target data structure of programs as t^{p}. Both patterns and programs are subsets of expressions. The expression t^{e} can be any term definable with the recursive $\mathbb{K}$ **IR** data structures described in Figure 3, while t^{p} is purely a ground.There are several constraints on t^{pat}, which will be described in later paragraphs. In any $\mathbb{K}$ **IR** rule, we also need to check that all metavariables in the right-hand side and the constrain are contained in the left-hand side, since only patterns can introduce metavariables. We translate `sortId` in `kItem` terms to a singleton set of the `sortId` `irKItem` terms.

Expression Translation Constraints. In translating expression from $\mathbb{K}$ **CORE** terms to $\mathbb{K}$ **IR** terms, several things need to be taken care of. First, we need to check if there are nested rewrites. In a $\mathbb{K}$ **CORE** term, if we see a sub-term whose constructor is **Rewrite**, and one of the constructor's arguments contains another **Rewrite** operator, the term is ill-formed. Second, datatypes of `k`, `aList`, `kList`, `bag`, `map` and `set`, which are originally in a binary tree form data structure in $\mathbb{K}$ **CORE**, need to be translated into the form of a cons list data structure such as the ones in (`irKElem list`), (`irListElem list`), (`irKListElem list`), (`irBagElem list`), (`irMapElem list`) and (`irSetElem list`). By this translation, we resolve the implicit identity or associativity equational rules in $\mathbb{K}$ **IR**, so that the sort system and pattern matching do not have to deal with identity or associativity equational rules.

Pattern Translation Constraints. In addition to the expression constraints, pattern translation has more restrictions. First, data structures of type (`irKElem list`), (`irListElem list`), (`irKListElem list`), (`irBagElem list`), (`irMapElem list`) or (`irSetElem list`), which are in cons list format, cannot have more than one element that is a metavariable. This restriction is a design choice of $\mathbb{K}$ since if there is more than one variable a an element in a list, then it is necessary for the $\mathbb{K}$ matching algorithm to consider all possible partitions of the list, which would create a significant loss of speed. Furthermore, we translate `set`, `bag`, `map` by purposely putting the metavariable element (if there is one) at the end position of the (`irSetElem list`), (`irBagElem list`) and

(`irMapElem list`) because it simplifies the pattern matching algorithm. We do not have a metavariable position restriction for translating `kList` and `aList`. The story of `k` is complicated. It is fine for users to write a term like: ($\rightsquigarrow$ (**Term** (**IdK** X)) ($\rightsquigarrow$ (**Term** (**IdK** Y)) (**Term** (**IdK** Z)))), where the three metavariable elements are `k` metavariables. All $\mathbb{K}$ implementations treat the variables X and Y as having sort `kItem` instead of `k` even though they have sort `k`, which is consistent with $\mathbb{K}$ matching algorithm description of the one variable requirement above. In order to translate this pattern, X and Y become the terms (**KElemIR** (**IdKItemIR** X `K`)) and (**KElemIR** (**IdKItemIR** Y `K`)), but Z becomes (**KIdIR** Z)))). Since **MacroRules** can switch positions of metavariables in an (`irKElem list`) term of an `irKRule`, we do not adjust the sorts here. Instead, we keep the sorts of X and Y until all **MacroRules** have finished causing specification changes.

Second, there is also a restriction on the occurrence of function `kLabel` names. $\mathbb{K}$ does not allow patterns containing function terms having the form `irKLabel` (`irKListElem list`), where the `irKLabel` is a function label, as flagged in the symbol table, in any **MacroRule**, **AnywhereRule**, **KNormalRule** or **BagNormalRule**. They only appear in the pattern of a **FunRule**. In the pattern of a **FunRule**, $\mathbb{K}$ only allows a function term to appear in the root position of a pattern term except the case when a function label as a `irKLabel` term appears as an element of an (`irKListElem list`).

Program Translation Constraints. By giving a configuration in a language specification and a program with some other user defined arguments, we generate an initial program state prototype that complies with the configuration of the specification. The result of replacing variables in the configuration by the input program and other possible argument should be a program state prototype without any metavariables in it. A check needs to be done to make sure of this. The program state prototype is not the initial program state because there are possible cells in the configurations whose **Multiplicity** feature is marked as $*$ or ?. $\mathbb{K}$ assumes that these cells are not presented initially but will be dynamically generated by later rules, except the cells initially containing `kCell` with the user input program. We need to eliminate all cells whose **Multiplicity** feature is marked as $*$ or ? from the program state prototype except those in the path from the top of the configuration to the initial `kCell`. By these processes, we generate the initial program state.

3.3 Translating $\mathbb{K}$ CORE rules

The process of translating $\mathbb{K}$ **CORE** rules to rules in $\mathbb{K}$ **IR** form is the implementation of the $trans^r$ function. The translation process is based on different kinds of rules that are divided according to different `synAttribs` or `ruleAttribs`. $\mathbb{K}$ **CORE** rules in general have the form: *RuleLabel LR C SL*, while $\mathbb{K}$ **IR** rules in general have the form: *RuleLabel L R C*, where *RuleLabel* is the header of a given rule, L is the pattern of the rule, R is the expression, LR is the first argument of a $\mathbb{K}$ **CORE** rule and has type `'a rewrite`, C is the condition and SL is the `synAttrib list`. The exception for **FunRule** is that it takes a list of pattern-expression pairs, instead of a single pattern and a single

Translating all different $\mathbb{K}$ **CORE** rules to $\mathbb{K}$ **IR** rules follows a similar process. First, it checks if a term involves **Rewrite** operators. In all cases except **Context** rules, this check rejects rules having nested **Rewrite** operators or no **Rewrite** operators. Second, we split the term LR by collecting all collecting all left hand sides of the **Rewrite** operators in the term to be the pattern and the right hand sides of the **Rewrite** operators to be the expression. Then, we translate the prototype pattern and expression as well as the condition term C to terms in $\mathbb{K}$ **IR** with checking these terms with other checks mentioned above. By using these terms as prototypes in $\mathbb{K}$ **IR**, we further adjust the translation according to different functionalities of rules below. Translating **Context** rules and bag rules requires modifications on the general patterns. For **Context** rules, we also need to locate *redex holes* in a given LR term and generate a fresh variable with a specific sort to cover the *redex holes* with other checks mentioned below. Translating bag rules require a complicated

process to finish splitting, "completing" and translating terms, which will be described in the later paragraph.

Generating FunRule, MacroRule, AnywhereRule and KNormalRule. $\mathbb{K}$ **CORE** and $\mathbb{K}$ **IR** have very different views of rules. $\mathbb{K}$ **CORE** distinguishes rules by types while $\mathbb{K}$ **IR** categories them by functionality. In $\mathbb{K}$, syntactic attributes have a higher priority than semantic attributes, so if a syntactic definition has a **Function** attribute in its (`synAttrib list`), and if the top most term of the pattern of a rule has the function label being defined by the function syntax, and a `ruleAttrib` of either **Macro** or **Anywhere**, then the `ruleAttrib` is ignored.

In order to build a **FunRule** for a function label from $\mathbb{K}$ **CORE** rules, we search all the rules to collect those whose the top most pattern terms have the given function label; these are the function rules defining the label. Then we translate the left hand sides of these $\mathbb{K}$ **CORE** rules into the patterns of an `irKLabel` (with exact the function label name) and an (`irKListElem list`), the right hand sides into expressions that have the same target sort as the sort in the symbol table entry for the function label, and the side condition into a `irKItem` term. If the two sorts are different, the language specification is rejected. We need to identify a rule containing the **Owise** attribute as the otherwise case for the function rules. We also need to check there are no two rules that have **Owise** attributes, which would cause the language specification to be rejected.

Translating a rule in $\mathbb{K}$ **CORE** into a **MacroRule** or **AnywhereRule** has a similar strategy to translating into **FunRules** but we are now dealing with a single rule instead of looking for a function label. All pattern checks need to apply on the pattern side of a given rule, and we distinguish a **MacroRule** or an **AnywhereRule** by looking at the `ruleAttrib` list, which must contain **Macro** or **Anywhere**, respectively. $\mathbb{K}$ does not forbid users to put **Macro** or **Anywhere** attributes on a rule whose left hand side cannot be translated into a pattern of an `irKLabel` and (`irKListElem list`), or whose right hand side cannot be translated to an (`irKElem list`) expression. In fact, $\mathbb{K}$ does not even forbid users from putting a condition on a **MacroRule**. Therefore such need to be checked and rejected to make sure that we are not translating them into $\mathbb{K}$ **IR** forms.

The only rules that can be translated into **KNormalRules** are **Context** rules and ones that have `kItem` or `k` terms and have no function label names, or **Macro** or **Anywhere** attributes. If a rule has only `kItem` as its pattern and expression, we generate a fresh variable at the end of the pattern and expression to upgrade the `kItem` term to a `k` term to represent the meaning of a computation sequence in a `kCell`. If there is a **Transition** attribute in the `ruleAttrib list` of the original rules, the **KNormalRules** indicate it in the final `bool` position.

When **Context** rules are translated into a pair of `heat/cool` **KNormalRules**, the process is similar to that for generating `heat/cool` pairs by **Strict** attributes, but we just need to check if all **Context** rules have exactly one *redex hole* if they do not have **Rewrite** operators in them, or, if they do have a **Rewrite** operator, then the left hand side of the **Rewrite** must be a *redex hole*, and the right hand side must be a `kItem` whose `kList` argument is a *redex hole*. When we translate a **Context** rule, we treat the pattern as that of the context rule with the condition `kItem` being translated, and add on the existing condition of generating heat/cool pairs.

Generating BagNormalRule. The most difficult part of the translation is the process of splitting, "completing" and translating `bag` data structures from $\mathbb{K}$ **CORE** to $\mathbb{K}$ **IR** since `bag` rules in $\mathbb{K}$ **CORE** cannot be simply translated by combining pattern and expression translation. Consider a `bag` rule from a language specification: ⬭ $u; fl$ (**Rewrite** A (`BagCon` A' B)), where u and fl are the cell name and feature list, A and A' are two cells that have a same cell name and B cell's **Multiplicity** feature is marked as $*$. If we split the term into a pattern and an expression directly, it looks like: `Rewrite` (⬭ $u; fl$ A) (⬭ $u; fl$ (`BagCon` A' B)). These two rules have different meanings. The

former means that we are not changing any contents in cell u and only changing cell A to cell A' and adding a cell B; the latter means that we are keeping only the contents of A in cell u, changing it to A', adding a cell B and changing all values of the other cells in sub-terms of cell u to the initial values defined by the configuration. To handle this subtly, we need to have a process that merges the three steps of splitting, "completing" and translating.

The main target of the merging step is the (`bag rewrite`) term of a `bag` rule. The input arguments of the merging step are the (`bag rewrite`) term and the configuration (in $\mathbb{K}$ **IR** format), as a prototype. The output of the merging step is a pair of (`irBagElem list`) terms acting as the pattern and the expression of a **BagNormalRule**, where the terms conform to the configuration. If we view the (`bag rewrite`) term and the configuration as two tree structures, then the merging step becomes the creation of two (`irBagElem list`) terms (two tree structures) containing information about the left hand and right hand sides of the (`bag rewrite`) term. Their nodes match the names and types of the configuration tree nodes and they preserve the configuration tree edges. The translation process becomes making sure the nodes and edges in the (`irBagElem list`) terms comply with the configuration. This is how we translate real (**Cell**) cells.

Dealing with a square cell is straightforward, because if we compile direct child cells of a square cell to an (`irBagElem list`) and locate the cells of same name in the configuration, these child cells must also be elements (direct children) of an (`irBagElem list`) term in the configuration whose name is the same as the square cell. They cannot appear as non-direct descendants of the the square cell. If there are metavariable child terms of the square cell, all other non-variable child cells of the square cell are in one-to-one correspondence with the elements of the (`irBagElem list`), and if there is not, the correspondence is bijective.

The hard part of translating a `bag` is dealing with a circle cell because the translation does not follow a simple pattern of mappin the children of the circle cell to their correct positions to preserve configuration information. If we find the sub-tree in the configuration whose root has the same name as the circle cell, the children of the cirle cell can appear in any descendants of the sub-tree as long as they are not nested. To solve this, we have an algorithm with input of the circle cell (a `bag` term) and the sub-tree (a (`irBagElem list`) term) and output a pair of (`irBagElem list`) terms as the pattern and expression of a **BagNormalRule**.

We introduce our algorithm with an inductive relation. The base case of the algorithm is that all the square cells and real cells can be taken care of as above. In the inductive step, we first assume that the algorithm can take care of all children cells of the input circle cell. Now, we divide the children into three categories: the first category is (`bag rewrite`) terms that have a top-most **Rewrite** operator. The second contains cells (singleton `bags`) that have **Rewrites** but they are not in the top-most position. The third category contains cells that have no **Rewrites** in them. For each element in the third category, we translate them directly to $\mathbb{K}$ **IR** form, fill the missing pieces with fresh variables having corresponding sorts with respect to the configuration, and create two copies of them: one as a pattern, the other as an expression. Elements in the second category must be child cells of the circle cell, so they will be taken care of by the inductive assumption.

For each term in the first category, since both the left hand side and the right hand side of the **Rewrite** term cannot contain any more **Rewrite** operators, we translate the left hand side and right hand side individually into a pattern and an expression (`irBagElem list`) term, respectively. If there are missing pieces in translating the left hand side, we fill them with fresh variables having the proper sorts. If there are missing pieces in translating the right hand side, we look at the configuration pattern and fill them with initial values from the configuration pattern.

After all three categories have been dealt with, we generate a list of child pattern cells and a list of expression cells in $\mathbb{K}$ **IR**. We compare the two lists with the input sub-tree configuration and put

them in the corresponding positions as in the configuration to create a pattern and an expression, and then fill the missing pieces with fresh variables having corresponding sorts for both the pattern and the expression. Then, we get the results as a pattern and an expression (`irBagElem list`).

After we translate the (`bag rewrite`) term of a bag rule and get a pattern and an expression term, the jobs left are to translate the condition term of the rule and perform some checks similar to those in translating **KNormalRules**, which are straightforward.

4 SORT SYSTEM FOR $\mathbb{K}$ IR

Previous materials about $\mathbb{K}$ only briefly describe the $\mathbb{K}$ sort system. The implementations of $\mathbb{K}$ have weak sort systems containing a lot of undesirable behaviors. In this paper, we 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 construct's argument sorts and 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 by subdividing different sort situations for a construct. They also want that when they run programs all the program states created sort-check. Besides these design goals, we also need to consider the language being sort-checked. The sort system is designed based on a $\mathbb{K}$ **IR** specification that has been translated from a $\mathbb{K}$ **CORE** specification through the compilation step, which makes the design of the sort system easier because we can assume that all (`irBagElem list`) terms should comply with the configuration. The sort system ensure this as well as that the sorts for the metavariables in a rule and sort restrictions for user defined constructs in a rule or a program state are applied properly.

Our sort system for $\mathbb{K}$ **IR** is order-sorted and partially ordered. The types `irKLabel`, (`irListElem list`), (`irSetElem list`), (`irMapElem list`), (`irBagElem list`) and (`irKListElem list`) are not allowed to have subsort relations and their sorts are unique. For all user defined constructs, we assume that they are subsorts of `KItem`, which is also a subsort of `K`. Users are allowed to define their own subsort relations on user created sorts and allowed to define subsorts of `KResult` and `KItem`. Once users define the subsorts of `KResult`, there is an implicit assumption that `KResult` is a subsort of `KItem`, `K` and any of the other user defined sorts. 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. In all later sections, we use $s, s_0, ..., s_n$ as sorts, and $S, S_0, ..., S_n$ as finite sets of sorts. We define a partial order relation $\leq$, where $s \leq s_0$ if s is a subsort of s_0. The function $\sqcup(S)$ computes the set of maximal lower bound sorts of an input sort set S. We use the notation $S \leq S_0$, and say S is a subsort of S_0, to mean that for every element s in S, we can find an element s_0 in S_0 such that $s \leq s_0$. Next, we describe the sort system and then introduce some important rules of the sort system.

Sort System to Generate Constraints. The $\mathbb{K}$ sort system is a sort refinement process to compute the greatest lower bounds of metavariables and constructs of a given rule or program state. It checks the sort a program state (sort checking) and adjusts the sorts of metavariables and constructs in a rule (sort adjustment) to be the set of maximal lower bound sorts under the input sort restrictions.

The $\mathbb{K}$ sort system is a binary relation in which the left side and right hand side are tuples of the form (α, β, S, t), where t is a term in $\mathbb{K}$ **IR**, α is a partial map giving maximal lower bound sorts for metavariables, and β is a partial maps to give maximal lower bound sorts for the terms labeled with `KLabel` metavariables. These `KLabel` variables are required to appear in the `irKLabel` position of an `irKItem` term in the pattern side. In $\mathbb{K}$, users are allowed to define an `irKItem` term with its label position being unspecified and written as a `KLabel` variable. The variable has sort `KLabel` in

an α map, but its instantiation also uniquely defines the target sort of the `irKItem` term if it is a non-function label, since a non-function label uniquely defines the least target sort of a construct based on the symbol table. A construct with a function label in $\mathbb{K}$ can produce other terms with non-function labels that have sorts being subsorts of the target sort of the function label, so it cannot guarantee the greatest lower bound of the construct. A construct with a function label can never appear in the pattern side of a rule, so requiring the domain of a β map to only include `KLabel` metavariables that at least appear once in the `irKLabel` position of a `irKItem` term in the pattern side can make sure that we are dealing with non-function label variables. A set of selected rules is described in Figure 4. For all situations except the rules dealing with (`irKListElem list`) terms, the input sort restriction is always a set of maximal lower bound sorts or $\top$, but the input sort restriction for (`irKListElem list`) terms is a list of sets of sorts as Sl, since (`irKListElem list`) represents the argument list and each position of it has a sort restriction; the output sort restriction is a set of sorts representing the maximal lower bound sorts for all situations.

$$\frac{\texttt{getKLabel}(X) = L \quad \texttt{isFunLabel}(L) \quad \texttt{sortOf}(L) = s \quad \texttt{argSortsOf}(L) = Sl \quad (\alpha, \beta, Sl, Y) \rhd (\alpha', \beta', \texttt{KList}, Y')}{(\alpha, \beta, S, \textbf{KItemIR}\ X\ Y\ S_0) \rhd (\alpha', \beta', \sqcup(\{s\} \cup S_0 \cup S)), \textbf{KItemIR}\ X\ Y'\ (\sqcup(\{s\} \cup S_0 \cup S)))}$$

$$\frac{\texttt{getKLabel}(X) = L \quad \neg\texttt{isFunLabel}(L) \quad \texttt{sortOf}(L) = s \quad \{s\} \leq S_0 \quad \{s\} \leq S \quad \texttt{argSortsOf}(L) = Sl \quad (\alpha, \beta, Sl, Y) \rhd (\alpha', \beta', \texttt{KList}, Y')}{(\alpha, \beta, S, \textbf{KItemIR}\ X\ Y\ S_0) \rhd (\alpha', \beta', \{s\}, \textbf{KItemIR}\ X\ Y'\ \{s\})}$$

$$\frac{\texttt{getKLabel}(X) = \bot \quad X \notin \texttt{dom}(\beta) \quad (\alpha, \beta, \texttt{KLabel}, X) \rhd (\alpha', \beta', \texttt{KLabel}, X') \quad (\alpha', \beta', \top, Y) \rhd (\alpha'', \beta'', \texttt{KList}, Y')}{(\alpha, \beta, S, \textbf{KItemIR}\ X'\ Y'\ S_0) \rhd (\alpha'', \beta'', \sqcup(S_0 \cup S, \textbf{KItemIR}\ X\ Y\ (\sqcup(S_0 \cup S))}$$

$$\frac{\texttt{getKLabel}(X) = \bot \quad X \in \texttt{dom}(\beta) \quad (\alpha, \beta, \top, Y) \rhd (\alpha', \beta', \texttt{KList}, Y')}{(\alpha, \beta, S, \textbf{KItemIR}\ X'\ Y'\ S_0) \rhd (\alpha', \texttt{add}(\beta', X, \sqcup(S_0 \cup S), \sqcup(S_0 \cup S, \textbf{KItemIR}\ X\ Y\ (\sqcup(S_0 \cup S))}$$

$$\frac{}{(\alpha, \beta, S, \textbf{HOLEIR}\ S_0) \rhd (\alpha, \beta, \sqcup(S_0 \cup S), \textbf{HOLEIR}\ (\sqcup(S_0 \cup S)))}$$

$$\frac{}{(\alpha, \beta, S, \textbf{IdKItemIR}\ X\ S_0) \rhd (\texttt{add}(\alpha, X, \sqcup(S_0 \cup S)), \beta, \sqcup(S_0 \cup S), \textbf{IdKItemIR}\ X\ (\sqcup(S_0 \cup S)))}$$

$$\frac{(\alpha, \beta, S, X) \rhd (\alpha', \beta', S', X') \quad (\alpha', \beta', Sl, tl) \rhd (\alpha'', \beta'', \texttt{KList}, tl')}{(\alpha, \beta, S\#Sl, (\textbf{KListElemIR}\ X)\#tl) \rhd (\alpha'', \beta'', \texttt{KList}, (\textbf{KListElemIR}\ X')\#tl')}$$

$$\frac{}{(\alpha, \beta, ([] \vee \top), []) \rhd (\alpha, \beta, \texttt{KList}, [])}$$

$$\frac{(\alpha, \beta, S, Y) \rhd (\alpha', \beta', S', Y') \quad (\alpha', \beta', Sl, (\textbf{KListIdIR}\ X)\#tl) \rhd (\alpha'', \beta'', \texttt{KList}, tl')}{(\alpha, \beta, Sl@[S], (\textbf{KListIdIR}\ X)\#tl@[\textbf{KListElemIR}\ Y]) \rhd (\alpha'', \beta'', \texttt{KList}, tl'@[\textbf{KListElemIR}\ Y'])}$$

$$\frac{}{(\alpha, \beta, Sl, [(\textbf{KListIdIR}\ X)]) \rhd (\texttt{add}(\alpha, X, \texttt{KList}), \beta, \texttt{KList}, [(\textbf{KListIdIR}\ X)])}$$

$$\frac{(\alpha, \beta, \top, (\textbf{KListIdIR}\ X)\#tl@[(\textbf{KListIdIR}\ Y)]) \rhd (\alpha', \beta', \texttt{KList}, tl')}{(\alpha, \beta, Sl@[S], (\textbf{KListIdIR}\ X)\#tl@[(\textbf{KListIdIR}\ Y)]) \rhd (\alpha', \beta', \texttt{KList}, tl')}$$

$$\frac{(\alpha, \beta, \top, tl) \rhd (\alpha', \beta', \texttt{KList})}{(\alpha, \beta, \top, (\textbf{KListIdIR}\ X)\#tl) \rhd (\texttt{add}(\alpha', X, \texttt{KList}), \beta', \texttt{KList}, (\textbf{KListIdIR}\ X)\#tl')}$$

$$\frac{(\alpha, \beta, \top, X) \rhd (\alpha', \beta', S', X') \quad (\alpha', \beta', \top, tl) \rhd (\alpha'', \beta'', \texttt{KList}, tl')}{(\alpha, \beta, \top, (\textbf{KListElemIR}\ X)\#tl) \rhd (\alpha'', \beta'', \texttt{KList}, (\textbf{KListElemIR}\ X')\#tl')}$$

$$\frac{\texttt{depends}(\mathit{RuleLabel})(S_0, S_1) \quad (\alpha, \beta, \texttt{topSort}(\mathit{RuleLabel}), L) \rhd (\alpha', \beta', S_0, L') \quad (\alpha', \beta', \texttt{topSort}(\mathit{RuleLabel}), R) \rhd (\alpha'', \beta'', S_1, R') \quad (\alpha'', \beta'', \texttt{bool}, C) \rhd (\alpha''', \beta''', \texttt{bool}, C')}{(\alpha, \beta, \mathit{RuleLabel}\ L\ R\ C) \rhd (\alpha''', \beta''', \mathit{RuleLabel}\ L'\ R'\ C')}$$

Fig. 4. Sort System For Generating Constraints (Selected Rules)

$\mathbb{K}$ allows users to write semantic rules in a flexible way. For example, users can write an **AnywhereRule** to rewrite a construct to a supersort construct. They can also define a metarule without specifying the actual `kLabel` for their constructs by using `KLabel` metavariables to represent them, as well as using a `KList` metavariable to represent part or all of the argument list of a construct. These features make it impossible to define a strong sort system for $\mathbb{K}$ like Standard ML's. Without the above situations, 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 these types of rules, by giving the assumption that $(\alpha, \beta, S, t) \rhd (\alpha', \beta', S_0, t_0)$, we have: (1) the output sort restriction is subsort to input sort restriction for sort checking and sort adjustment. (2) if we apply sort checking and sort adjustment twice, the first output sort restriction and the second one will be the same. (3) there is a substitution φ whose domain is the domain of α such that, if $\varphi(X)$ sort-checks by the sort restriction defined by $\alpha'(X)$ for any X in the domain of

α and $\varphi(t)$ sort-checks by the sort restriction defined by $\beta'(X)$ for any `kItem` t having `irKLabel` position as metavariables in the domain of β, then for any term $\varphi(t')$ that sort-checks, the output sort restriction of $\varphi(t')$ is subsort of the input sort restriction $\varphi(t')$.

Some Sort System Rules. The sort system described in the Figure 4 is a prototype leading to satisfy our $\mathbb{K}$ sort system design goals and guarantees. While we are showing only sort adjustment rules here, the sort-checking rules are a simplified version of these rules by cutting the α and β transitions from the sort adjustment rules.

In describing the sort system, we use a number of auxiliary functions. The function `dom()` gives the domain of a map. The partial function `getKLabel` gives the fixed label name of an `irKLabel` term with a fixed label. The function `isFunLabel` checks if a label is a function label or a normal label (non-functional label). The function `sortOf` gives the sort of a label name, while `argSortsOf` gives argument sorts for a label name. The function `add` adds a sort constraint to the sort list of a metavariable and calculates the new maximal upper bounds of the given sorts. The function `topSort` finds the top most sort of a term according to different *RuleLabels*. For example, the **FunRules** with `irKLabel` terms are given a top sort of `KLabel`, **AnywhereRules** are given top sort of `K` and **BagNormalRules** are given top sort of `Bag`. Finally, `depends` is a function that gives different checks depending on the *RuleLabels*. If we are dealing with a **FunRule**, the left hand side sorts S and right hand side sorts S_0 should be subsorts of the target sort defined by the `irKLabel` of the **FunRule**. If we are dealing with a **MacroRule** or an **AnywhereRule**, S and S_1 should both be subsorts of {`K`}. For a **KNormalRule** or a **BagNormalRule**, S and S_1 should both be {`K`} or {`Bag`}, respectively.

We only show rules for `irKItem` and (`irKListElem list`), since other rules are simplified versions of theses, or rules that union information from recursive calls. In dealing with an `irKItem` pattern, we divide the `irKLabel` (`irKListElem list`) s pattern into four different cases. First, if the `irKLabel` piece introduces a function label, we calculate the maximal lower bounds of the input restriction, a sort associated with the term and the singleton set containing target sort introduced by the function label. If the `irKLabel` piece introduces a normal label, we need to check if the target sort introduced by the label is a subsort of the sort associated with the term, its singleton set is a subsort of the input restriction. If the `irKLabel` piece does not introduce any labels, and it is not a variable in the domain of β, we need to calculate the maximal lower bounds of the input restriction and the `irKItem` associated sort. Finally, if the `irKLabel` piece is a metavariable in the domain of β, we need to add the constraint sort to β for the variable by calculating the maximal lower bounds of the input restriction and the `irKItem` associated sort.

$\mathbb{K}$ allows metarules involving terms having `KList` variables that represent a list of arguments, and evaluation of these rules can take place if a list of arguments with correct sorts are provided at runtime. In dealing with this case, we divide the sort-checking process of (`irKListElem list`) into two different cases: an existing input restriction list of sorts and non-existing input restrictions. If there is an input restriction sort list, we start from the two ends of the input (`irKListElem list`). If the element in either of the two ends are not `KList` metavariables, then we match the element with the corresponding position sort in the input restriction sort list. If the two ends are `KList` metavariables, we remove the input restriction list or do the rest of the sort-check the same as a non-existing input restrictions check. If there is no input restriction list, we just traverse the (`irKListElem list`) and sort-check each individual list item. Finally, for each rule pattern *RuleLabel* L R C, we need to sort adjust L, R and C based on the generic rule described in the figure.

Normalizing Terms. Sort adjustment means that using the information from the two mapping functions generated from sort adjusting, we adjust the sorts for each metavariable and `irKLabel`

(irKListElem list) S term with the values of the mapping functions. Normalization is a process happens after sort adjustment, both being steps in $\mathbb{K}$_compile, to apply idempotent and functional equational rules to any sub-terms of rules and program states having the types (irSetElem list) or (irMapElem list). The process is to get rid of redundant child elements of (irSetElem list) or (irMapElem list) sub-terms and make sure (irMapElem list) is functional. Normalization also happens after sort checking in doing a pattern matching in the evaluation step.

5 SEMANTIC DEFINITION OF $\mathbb{K}$ IR

In defining $\mathbb{K}$ **IR** rule behaviors, the input is a $\mathbb{K}$ theory from which we can extract information to form lists of **MacroRules**, **FunRules**, **AnywhereRules**, **KNormalRules** and **BagNormalRules** in $\mathbb{K}$ **IR** form by the compilation step. A program is provided for the language specification defined by the $\mathbb{K}$ theory, and it is compiled with the configuration to an initial program state as an (irBagElem list) in $\mathbb{K}$ **IR**. There is also a number n that is either a natural number or a $\top$ to represent the steps of evaluation. If n is a $\top$, it means that users want to obtain the final result of a given initial program state by applying rules to the program state as many times as they can. In $\mathbb{K}$, only the application of a **KNormalRule** or **BagNormalRule** to a program state is recognized as a step. An order function is also give for every rule list above to define the order of the list for picking a rule to evaluate. If a user does not use $\mathbb{K}$_search, and there is more than one rule that can be applied to a given program state, the rule that has the highest order is selected. The current $\mathbb{K}$ implementation uses the lexicographic order to decide which rule to apply at a given step.

The basis for defining $\mathbb{K}$ **IR** rule behaviors is the pattern matching algorithm, which takes into account equational rules on $\mathbb{K}$ **IR** built-in terms and the pattern matching of a metavariable not only against a ground term (for dealing with a **FunRule**, **AnywhereRule**, a **KNormalRule** or a **BagNormalRule**) but also against a term involving metavariables (for dealing with a **MacroRule**). The behaviors of $\mathbb{K}$ **IR** rules can be divided into two kinds. The first is preprocessing, which invloves applying all the **MacroRules** to all the rules, and then performing well-formedness and sort checks on the result. The second is evaluation, which begins with applying function rules and **AnywhereRule**s. We first apply any function rule to the program state where the function label occurs on a subterm that is not a direct element of an (irKListElem list) subterm. Once the program state has no function labels to be rewritten, we apply a possible **AnywhereRule** to it. After the **AnywhereRule** have been applied, we return function rule application before applying the next possible **AnywhereRule**. When no **AnywhereRules** or function rules apply, we finish the evaluation step by applying a rule in the lists of **KNormalRules** and **BagNormalRules** if there are any applicable.

5.1 Pattern Matching Algorithm

The pattern matching algorithm is a combination of a pattern matching step, substitution step, sort adjusting step and normalization step, since we need to make sure a system is dynamically well typed after it transits through a rule. The pattern matching step discovers a mapping function from a set of metavariables to ground or non-ground terms. The substitution step takes the mapping function and substitutes all of its keys for the values in a given program state. Next we need to use a sort adjusting/checking function derived from the sort system, as described in Section 4, to force and adjust the sorts in a term; then we need to normalize the term as described in Section 4.

The substitution, sort adjusting/checking and normalization steps are either trivial or described in the previous section. Here, we focus on pattern matching step. The assumption to implement it is that all input terms should be sort adjusted and normalized. The pattern matching step usually means that for a given pattern P and term t, if there is a mapping function m from some

metavariables to terms such that $m(P) = t$. The $\mathbb{K}$ pattern matching step has three differences from other pattern matching algorithms.

First, $\mathbb{K}$ has a sort system that involves subsorts such that the pattern P matches a term t if a mapping function exists and the target sort of P is a supersort of the target sort of t. Second, **MacroRules** require that the $\mathbb{K}$ pattern matching step not only deal with ground terms but also with term involving metavariables. In this case, if the pattern P is a metavariable with sort S, and t is a metavariable with sort S', then we can have a map by mapping P to t, if $S' \leq S$. One thing to keep in mind is that the algorithm of $\mathbb{K}$ is a pattern matching algorithm so if P is a term (not a metavariable), and t is a metavariable, it means that the pattern matching fails instead of t mapping to P.

Third, because of the equational rules on $\mathbb{K}$ **IR** built-in terms, we also need to deal with non-linear commutativity in (`irBagElem list`), (`irMapElem list`) and (`irSetElem list`), since $\mathbb{K}$'s restricted pattern formats make identity, associativity, idempotence and being functional equational rules easily be taken care of by the compilation, type adjustment and normalization steps described above. Dealing with commutativity means that, given a pattern P in (`irBagElem list`), (`irMapElem list`) or (`irSetElem list`), and a term t in (`irBagElem list`), (`irMapElem list`) or (`irSetElem list`), if there is a mapping function m, for every element P_i in P, there is an element t_j in t, such that $m(P_i) = t_j$. Commutativity pattern matching is NP-complete. The commutativity pattern matching step is basically to find if the mapping function m can be bijective. To make the pattern matching step a little efficient in most cases, we first collect all possible term elements being pattern matched with a pattern element P_i and do this for every pattern element. Then, we pick a pattern element P_i having smallest number of possible matched term elements, and pick a term element t_j, then remove all occurrence of t_j in all other pattern elements. By doing this process recursively, if we find there is a pattern P_k having empty matching set, which means that we cannot have a bijection mapping, we recurse to pick another choice until we find the mapping or try all cases and fail. The process can make most pattern matching cases efficient because the chance of having all pattern elements that have a lot of different possible matched term elements with all most matching cases being invalid is very small.

5.2 Preprocessing Step and Final Compilation Check

Dealing with **MacroRules** is one of the hardest jobs in defining $\mathbb{K}$ semantics since there is no source which talks about the proper behaviors of **MacroRules**. Furthermore, running **MacroRules** in $\mathbb{K}$ 3.6 and 4.0 results in a lot of undesirable behaviors. It is necessary to put restrictions on **MacroRules** to make it behaves consistently. First, we require the pattern side of a **MacroRule** to live in a type `irKItem` and the expression side of a **MacroRule** to live in a type (`irKElem list`). Second, we require the pattern of a **MacroRule** to be in the form `irKLabel` (`irKListElem list`) s so that it cannot rewrite an `irKItem` metavariable or a *redex hole* since these rewritings would make no sense. Third, we require **MacroRules** to have no conditions, since a condition require dynamic evaluation, and applying **MacroRules** is a preprocessing step that should not involve dynamic evaluation. Fourth, we require the `irKLabel` of a **MacroRule** pattern side not to appear in the expression of the **MacroRule**, since the main purpose of having **MacroRules** is to allow desugaring, and having an `irKLabel` on the expression allows the possibility of recursions and a non-terminating preprocessing step.

The semantics of **MacroRules** has only one rule, as in Figure 5. In this rule, *Ml* and *Rl* represent ordered **MacroRules** and other rules in a language specification, respectively. C represents the initial program state. `macroMatch`(r, Ml) is a partial function implementing the pattern algorithm described above. It first takes a rule or program state term, searches its sub pieces, matches the

$$\frac{\texttt{macroMatch}(r, Ml) = Ml' \quad \texttt{macroMatch}(r, Rl) = Rl' \quad \texttt{macroMatch}(r, C) = C' \quad \texttt{validMacroRules}(Ml')}{\texttt{macro}\ (r\#Ml, Rl, C) \longrightarrow (Ml', Rl', C')}$$

$$\frac{\texttt{hasFunTerm}(C) \quad r \in Fl \quad \texttt{funMatch}(r, C) = C'}{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{f}} (Fl, Al, Kl, Bl, C')}$$

$$\frac{\neg\texttt{hasFunTerm}(C) \quad r \in Al \quad \texttt{anywhereMatch}(r, C) = C'}{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{a}} (Fl, Al, Kl, Bl, C')}$$

$$\frac{\neg\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{a}} (Fl, Al, Kl, Bl, C') \quad \neg\texttt{hasFunTerm}(C) \quad r \in Kl \quad \texttt{kNormMatch}(r, C) = C''}{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{o}} (Fl, Al, Kl, Bl, C'')}$$

$$\frac{\neg\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{a}} (Fl, Al, Kl, Bl, C') \quad \neg\texttt{hasFunTerm}(C) \quad r \in Bl \quad \texttt{bagNormMatch}(r, C) = C''}{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{o}} (Fl, Al, Kl, Bl, C'')}$$

$$\frac{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{f}}{}^{*} (Fl, Al, Kl, Bl, C') \quad \texttt{core}\ (Fl, Al, Kl, Bl, C') \xrightarrow{\texttt{a}} (Fl, Al, Kl, Bl, C'')}{\texttt{tauStep}\ (Fl, Al, Kl, Bl, C) \longrightarrow (Fl, Al, Kl, Bl, C'')}$$

$$\frac{n > 0 \quad \texttt{tauStep}\ (Fl, Al, Kl, Bl, C) \longrightarrow^{*} (Fl, Al, Kl, Bl, C') \quad \texttt{core}\ (Fl, Al, Kl, Bl, C') \xrightarrow{\texttt{o}} (Fl, Al, Kl, Bl, C'')}{\texttt{oneStep}\ (n, Fl, Al, Kl, Bl, C) \longrightarrow (n-1, Fl, Al, Kl, Bl, C')}$$

$$\frac{\texttt{core}\ (Fl, Al, Kl, Bl, C) \xrightarrow{\texttt{f}}{}^{*} (Fl, Al, Kl, Bl, C') \quad \neg\texttt{core}\ (Fl, Al, Kl, Bl, C') \xrightarrow{\texttt{a}} (Fl, Al, Kl, Bl, C'')}{\texttt{tauStep}\ (Fl, Al, Kl, Bl, C) \longrightarrow (Fl, Al, Kl, Bl, C')}$$

$$\frac{\begin{array}{c}\texttt{compile}(Spec) = (Ml, Fl, Al, Kl, Bl, CP) \quad \texttt{sortAdjust}(Rl) = Rl' \\ \texttt{compile}(CP, p) = C \quad \texttt{split}(Rl') = (Fl', Al', Kl', Bl') \\ \texttt{macro}\ (Ml, Fl@Al@Kl@Bl, C) \longrightarrow^{*} ([], Rl, C') \\ \texttt{oneStep}\ (n, \texttt{funs}@Fl', Al', Kl', Bl', C') \longrightarrow^{*} (m, Fl', Al', Kl', Bl', C'')\end{array}}{\mathbb{K}\texttt{\_run}\ (Spec, n, p) \longrightarrow C''}$$

Fig. 5. $\mathbb{K}$ **IR** Semantics

pattern in r with these sub-terms and substitutes the sub-term that successfully matches the expression of r and fills in the pattern matching mapping function generated. We also need to do a sort adjustment and normalization step after this. `validMacroRules` is a function to see if a list of **MacroRules** is valid according to the above restrictions.

After we finish the preprocessing step, we need to complete the final step of $\mathbb{K}$`_compile` and then move to evaluate the program states. The function `sortAdjust` is the final compilation step for adjusting the sorts in rules. Recall from Section 3, we have X and Y being type `kItem` instead of `k` in the example: (⇝ (`Term` (`IdK` X)) (⇝ (`Term` (`IdK` Y)) (`Term` (`IdK` Z)))) due to the limitations of the pattern matching algorithm in $\mathbb{K}$. After translating into $\mathbb{K}$ **IR**, the metavariables X and Y become (`irKElem list`) terms, and we need to change the terms of X and Y to `IRIdKItem` X `KItem` and `IRIdKItem` Y `KItem`, respectively, since they should be in sort `KItem` instead of `K`. The job of `sortAdjust` is to apply this adjustment process to every sub-terms in patterns of all rules, and then running type adjustment to properly inferring the sorts of terms. The reason we need to do this adjustment here is because a **MacroRule** can switch the element positions of an (`irKElem list`) term. For example, after we apply certain **MacroRules**, the example above can become (⇝ (`Term` (`IdK` Z)) (⇝ (`Term` (`IdK` X)) (`Term` (`IdK` Y)))). We can see there is certain oddness in the design of $\mathbb{K}$ due to **MacroRules** and we will talk about this in a later section.

5.3 Evaluation Step

The evaluation step in this section describes the behavior of $\mathbb{K}$`_run`. For a given program state, we apply **FunRules**,**AnywhereRules**, **KNormalRules** or **BagNormalRules** and get a final program state that either applies several steps of normal rules or freezes up completely.

The rules are given in Figure 5. In this figure, Fl, Al, Kl and Bl represent lists of **FunRules**, **AnywhereRules**, **KNormalRules** and **BagNormalRules**, respectively. r is a single rule, CP is a configuration definition in a $\mathbb{K}$ specification p is a given program, C, C' and C'' are program states and n and m are natural numbers representing steps. We describe the $\mathbb{K}$`_run` by a label transition system such that every transition rule contains a rule label including `core`, `tauStep`, `oneStep` and $\mathbb{K}$`_run`, describing the behavior of the rule. The left hand side and right hand side are transition tuples that typically contain rules and a program state. here may be one of three labels over the arrow to indicate the type of transition: `f`, `a` and `o`, which represent **FunRule**, **AnywhereRule** and normal rule transition, respectively.

In these rules, `funMatch`, `anywhereMatch`, `kNormMatch` and `bagNormMatch` are functions similar to `macroMatch`, and represent a pattern matching procedure. The only difference among these functions is where they do the pattern matching. `funMatch` looks at the sub-terms of a given program state, finds a term with a function label and does pattern matching and substitution steps on it. `anywhereMatch` looks at the sub-terms of a given program state, finds a term corresponding to the pattern side of a given **AnywhereRule** and does pattern matching and substitution steps on it. `kNormMatch` looks only at the `kCells` of a given program state, locates a `kCell` corresponding to the pattern side of a given **KNormalRule** and does pattern matching and substitution steps on the (`irKElem list`) term in the cell. `bagNormMatch` does pattern matching and substitution steps on the whole program state. `hasFunTerm` is a function to see if a term has sub-terms with a function label. `compile` is a function to finish jobs in the compilation step, while `split` is a function to divide a set of rules into four lists: **FunRules**, **AnywhereRules**, **KNormalRules** and **BagNormalRules**.

$\longrightarrow^*$ means multiple steps of transition typically reaching a state that cannot move anymore. For example, in the $\mathbb{K}$_`run` rule, the `oneStep` deduction step has an $\longrightarrow^*$ that means that either the program state C'' on the right hand side cannot make any more moves or that the step indicator m becomes zero. All rules in Figure 5 should be very straight forward, except that only a step caused by **KNormalRules** or **BagNormalRules** is recognized as a normal step, triggering the step indicator to decrease. In addition, in $\mathbb{K}$, we can only start using **AnywhereRules** and normal rules to evaluate a program state, if a program state is without any function label sub-terms. The condition to trigger an **AnywhereRule** and normal rule evaluation is the given program state – it should not have function label terms; while the way to keep **AnywhereRule** from applying is to disprove the existence of transitions with the label `a`.

5.4 $\mathbb{K}$_search

$\mathbb{K}$_`search` is the trace version of $\mathbb{K}$_`run`. The difference between $\mathbb{K}$_`search` and $\mathbb{K}$_`run` is that $\mathbb{K}$_`search` allows users to see all possible traces while $\mathbb{K}$_`run` only allows users to see an evaluation sequence. $\mathbb{K}$_`search` relies completely on users defining the rules designed to represent non-deterministic choices with the **Transition** attribute. Only **KNormalRules** and **BagNormalRules** labeled with the **Transition** attribute can observe of these non-deterministic behaviors, which is why only **KNormalRule** and **BagNormalRule** constructs having a boolean argument in the end indicating if the rule is labeled with a **Transition** in the `irKRule` category.

The $\mathbb{K}$ documents do not specify the details of $\mathbb{K}$_`search`, but both $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0 design $\mathbb{K}$_`search` in the same way. For a given program state in a specification, $\mathbb{K}$_`search` 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 capably applying to the program state, $\mathbb{K}$ starts looking at the **Transition** rules, and collects a set of results by applying all of the rules to the program state. The implementation of $\mathbb{K}$_`search` is similar to the $\mathbb{K}$_`run` 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 $\mathbb{K}$_`run`; if not, we need to apply all of the **Transition** rules to the program state and to get the results.

6 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 description of $\mathbb{K}$ 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.

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 the major features of $\mathbb{K}$. It also has 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 the grammar and semantics of it. In this sense, these $\mathbb{K}$ specifications are far from being formal.

Matching Logic is a logic system that is built on top of $\mathbb{K}$ to reason 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 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 indicate the a logic proof system is the future that the $\mathbb{K}$ project is pursuing.

Other work in $\mathbb{K}$ [Arusoaie et al. 2012; Şerbanuţă and Roşu 2010; Ştefănescu et al. 2016; Ellison et al. 2009; Hills et al. 2008; Hills and Roşu 2008; Lucanu et al. 2012; Meseguer and Roşu 2011; Şerbănuţă et al. 2009] talks about other useful compilations $\mathbb{K}$ 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.

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 exectuable 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 by Park and Rosu [Bogdănaş and Roşu 2015]. They were both tested by a 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 that is designed to define other language specifications, which means that we need to define both the compilation and executions of $\mathbb{K}$, which we did in defining $\mathbb{K}$_compile, $\mathbb{K}$_run and $\mathbb{K}$_search. Second, we also did a test bank to test our $\mathbb{K}$ specification but we can communicate constantly with the $\mathbb{K}$ group members to ascertain what they are thinking 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, we built our specification in Isabelle/HOL [Nipkow et al. 2002], which is more difficult and trust worthy than other specifications in $\mathbb{K}$ based on Java. We also have an ongoing proof in Isabelle/HOL to prove the soundness and completeness of our $\mathbb{K}$ specification with a translation of the $\mathbb{K}$ specification into Isabelle/HOL.

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 Maharaj and Gunter [Maharaj and Gunter 1994]. In contrast to ML, formalizing real world langauge 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 a significant portion of C. They used Coq to generate compiled code behaving exactly as described by the specification of the language. Even though the use of 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 is not extracted from Coq, although it would be possible to obtain an interpreter without too much additional effort. Other projects based on CompCert include AppelâĂŹs, 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 is one of the purposes for designing $\mathbb{K}$. Compared to Ott, $\mathbb{K}$ is designed as a programming language to allow users to reason about language specifications by $\mathbb{K}$ 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#: EisenbachâĂŹs formal Java semantics [Drossopoulou et al. 1999] and SymeâĂŹs HOL semantics [Syme 1999] of

Drossopoulou; the C# standard by BÃűrger 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. 2013, 2011] is another important work having a real impact on the C11 standard.

Our mechanized specifications of $\mathbb{K}$ 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 $\mathbb{K}$. They are detailed in previous sections.

7 EVALUATION, APPLICATION AND CONCLUSION

The process of evaluating $\mathbb{K}$ semantics is the process of building trust in it with respect to real $\mathbb{K}$. We talked to the $\mathbb{K}$ team in depth. In the first several months of our $\mathbb{K}$ semantics project, we only did multiple cycles of (1) discussing materials with the $\mathbb{K}$ team, (2) implementing critical thoughts as some small language specifications and running them in the $\mathbb{K}$ implementations, then (3) discussing more materials with them. By gathering thoughts from the $\mathbb{K}$ team, we can conclude that we have the best knowledge about $\mathbb{K}$ outside of the $\mathbb{K}$ team. Then, we started to define $\mathbb{K}$ semantics. In doing so, we still consulted with the $\mathbb{K}$ team to make sure our $\mathbb{K}$ semantics corresponded with what they were looking for. The second step was to run test cases against our $\mathbb{K}$ semantics and compare the results with current $\mathbb{K}$ implementations, especially $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0. The interpreter we built was directly extracted from our theory files in Isabelle. We did this extraction in Isabelle/HOL 2016. We ran 356 programs in total for 13 different language specifications, and our results showed that our $\mathbb{K}$ interpreter passed 338 programs. The specifications and programs are the test bank that the $\mathbb{K}$ team is using to test their $\mathbb{K}$ implementations for both $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0. The role of the test bank is very similar to the GCC torture test suite for testing the C specification. All results are listed in the `k-tests` folder.

Among the test cases, we had no single specification that we cannot handle at all. Our $\mathbb{K}$`_compile` function compiled all test specifications, but there are test programs that we cannot handle. All of them relate to the standard input channel. Recall that $\mathbb{K}$ allows users to define features for a cell, one of which is **Stream**, whose **In** keyword allows users to type in inputs on a keyboard and then put the results in the input cell of a given $\mathbb{K}$ program state, just as C and Java do. This is hard to do in Isabelle directly, and the best way to handle it is to assume there is an input cell which generates some inputs. However, for users to really type in inputs during testing, the code of our $\mathbb{K}$ interpreter needs to be changed. At the time of this paper's submission, we have not yet finished the job, but we believe that it is an easy fix for our interpreter.

We also build the trust by our further study on the project. There is an ongoing effort to translate language specifications in $\mathbb{K}$ to language specifications in Isabelle, and to prove the translation bi-simulates any language specifications based on the $\mathbb{K}$ semantics in Isabelle and their translations in Isabelle. The bi-simulation also proves the soundness and completeness of our $\mathbb{K}$ semantics.

7.1 Different Applications and Weakness in $\mathbb{K}$

Defining semantics for a language always has a lot of benefits. This is a statement that has been proved by many previous projects defining semantics, such as those described in Section 6. Unlike other languages, the purpose of $\mathbb{K}$ is to define other language specifications, which makes its semantics more useful than other language specifications. While more imaginative people might come up with more, we propose a few usages here., we propose a few usages here.

Discovering Undesirable Behaviors. Formal language specifications are widely used to discover undesirable behaviors in the implementation of a language. Undesirable behaviors can either be a difference as in the language specification document, or in something that is not described in

any document but can be inferred, either by the design patterns of the language, or by the language designers. Undesirable behaviors can be very annoying to the users of the language because they are shocking aberrations from the expected performance, and need a lot of time to figure out. They usually end up wasting a lot of the users' time, which is a lot more costly than running a better, more efficient implementation of a program. This is why the most important job in programming language field is to make sure language specifications are correct.

If the $\mathbb{K}$ implementations exhibit undesirable behaviors, it is specially bad because people use $\mathbb{K}$ to define their language specifications. Once an undesirable behavior occurs, it is hard for them to distinguish if it is from the design of their target language or $\mathbb{K}$. By running test specifications and comparing the results with $\mathbb{K}$ 3.6 and $\mathbb{K}$ 4.0, we identified 25 kinds of undesirable behavior in $\mathbb{K}$. Each can have many different versions but we only show one version for each kind in our folder named `undesirable-behaviors`. All these test specifications have a unique input program so users can see them easily. Only two or three of them are related solely to just $\mathbb{K}$ 3.6 or $\mathbb{K}$ 4.0; the others are found in both implementations.

These undesirable behaviors happen in very diverse circumstances. For example, some are related to the sort checking. Current $\mathbb{K}$ implementations allow users to set up rules to rewrite a `k` term to an `aList` or `set` term, which indicates that the sort checking currently implemented does not guarantee a lot, and makes $\mathbb{K}$ more like an unsorted algebra. Second, some undesirable behaviors are related to `cells` in $\mathbb{K}$. By not specifying a **Multiplicity** feature in the configuration, the current implementation allows users to remove or add a `cell` as long as the `cell`'s name is in the configuration. This makes it harder for the design of the sort system to guarantee the correctness of the program state, and makes the **Multiplicity** features vulnerable. Third, some undesirable behaviors are related to the pattern matching algorithm in $\mathbb{K}$. The current $\mathbb{K}$ implementations allow some associative and identity implicit equational rules for user defined list operators in a language specification defined by the constructor **ListSyntax**. However, there are some cases where the associative rewriting does not work, which is why we decided to not allow associative and identity implicit equational rules for user defined list operators. In addition, the implementation of the commutative implicit equational rule also fails in some cases. Due to the page limitation on the paper, we cannot list all of the kinds of undesirable behaviors here. However, there are clearly undesirable behaviors in $\mathbb{K}$.

Suggesting Design Changes in $\mathbb{K}$. Defining the formal semantics for a language suggests design patterns to language developers. The $\mathbb{K}$ sort system defined in this paper and the pattern matching algorithm are patterns the $\mathbb{K}$ team can use in their $\mathbb{K}$ implementations.

Moreover, from our experience in defining $\mathbb{K}$ semantics, we suggest some design changes for the $\mathbb{K}$ team. First, **MacroRules are harmful and useless**. About four of the undesirable behaviors are related to **MacroRules**. When we deal with **MacroRules** in defining the semantics, we must add a lot of restrictions and checks on them to make them right. We even need to change the pattern matching algorithm to allow pattern matching against expressions including metavariables because **MacroRules** is applicable to the language specification itself. From the example in Section 5, we can see that **MacroRules** can affect the meaning of a rule involving a `k` term due to the limitation that the associative pattern matching algorithm $\mathbb{K}$ is providing (because the $\mathbb{K}$ team wants to make the algorithm much faster). By glancing at the different previous language specifications in $\mathbb{K}$ [Bogdănaş and Roşu 2015; Ellison and Rosu 2012; Filaretti and Maffeis 2014; Park et al. 2015] and the $\mathbb{K}$ test suite, all uses of **MacroRules** are translating one operator into another, which can easily be achieved by defining the target operator as a function term and then using the function rules. Hence, **MacroRules** are necessary in $\mathbb{K}$.

Second, **Avoid AnywhereRules** They are somewhat harmful to the sort system of $\mathbb{K}$. Actually, if there are no **AnywhereRules** or normal rules involving metavariables for `kLabel` in a specification, the sort system defined in this paper can have a much stronger guarantee that, once the specification and the program state are sort checked, there is no way that executing the program state is ill sorted. In addition, one of the future goals of the $\mathbb{K}$ project and our goal for the $\mathbb{K}$ semantics in Isabelle is to have an interactive theorem prover based on $\mathbb{K}$. Imagine this: in order ro apply a normal rule to a program state to make a move, one needs to prove there are no **AnywhereRules** to apply first. This requires an induction proof on the set of **AnywhereRules** against the program state, which is a lot of work to do for every move of a system. Certainly, one can change the $\mathbb{K}$ semantics a little to make the condition of "no **AnywhereRules** to apply" mean the same as "program state does not changing after applying **AnywhereRules** ". In some cases, however, we just cannot define "no change of a program state" to mean "there are no rules to apply".

Model Checking. All previous major specifications in $\mathbb{K}$ have claimed some verification tools in their semantics. By examining these executable specifications in $\mathbb{K}$, we find that all of the verification tools rely on changing or extending their specifications to include the specifications of the tools, which is not an ideal case for getting a verification tool.

Based on our $\mathbb{K}$ semantics, we can build verification tools on top of them, and these tools are language independent and beneficial to all specifications defined in $\mathbb{K}$. We show an implementation of the CTL model checker here, but we believe that it is easy to follow the same design pattern to make an LTL model checker based on our $\mathbb{K}$ semantics, as well as other verification tools.

The CTL model checker in $\mathbb{K}$ includes all major operators of CTL with an extra functions. The extra function asks users to input a list of `cell` names refering to the `cells` in a program state that users care about. The basic predicates in the CTL formula allow users to write a list of pairs of a `cellId` and a pattern refering to what users consider as valid in a given program state. Using the function and the CTL formulas, we use the same mechanism as $\mathbb{K}$`_search` to generate traces in a form of a finite automaton structure by merging the same program states in the traces. If the program states are infinite, then the model checker fails. After generating the automaton structure, we use the traditional CTL model checking algorithm of Clarke and Emerson [Clarke and Emerson 1982] to model check it.

7.2 Conclusion

In this paper, we proposed three different syntaxes for describing $\mathbb{K}$, showed how to translate $\mathbb{K}$ **CORE** into $\mathbb{K}$ **IR**, suggested a sort system, and pattern matching algorithm for K, and defined the semantics of $\mathbb{K}$ based on the $\mathbb{K}$ **IR** language. All of these processes involved discussion with the $\mathbb{K}$ team to make sure our $\mathbb{K}$ formal semantics behaved correctly. We also examined our $\mathbb{K}$ semantics by running tests against our system and found that our system passed all 13 test specifications and 338 out of 356 programs for these test specifications. We also discovered 25 major undesirable behaviors of $\mathbb{K}$ and suggested some design changes to the $\mathbb{K}$ team.

Furthermore, we also show how to build verification tools on top of our $\mathbb{K}$ semantics by an example of a generic and language independent CTL model checker. This is one of the paths that we will follow in our future work on the $\mathbb{K}$ semantics project. Another path we would like to pursue is to marry the transitional interactive theorem prover field with the $\mathbb{K}$ group by translating specifications in $\mathbb{K}$ to ones in Isabelle/HOL. This will not only improve the correctness of our $\mathbb{K}$ semantics but also allow researchers to prove properties about large language specifications in $\mathbb{K}$ [Bogdănaş and Roşu 2015; Ellison and Rosu 2012; Filaretti and Maffeis 2014; Park et al. 2015] that may not exist in Isabelle/HOL.

REFERENCES

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

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

Andrei Arusoaie, Traian Florin Şerbănuţă, Chucky Ellison, and Grigore Roşu. 2012. Making Maude Definitions more Interactive. In *Rewriting Logic and Its Applications, WRLA 2012 (Lecture Notes in Computer Science)*, Vol. 7571. Springer.

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

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

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2006. Engineering with Logic: HOL Specification and Symbolic-evaluation Testing for TCP Implementations. *SIGPLAN Not.* 41, 1 (Jan. 2006), 55–66. https://doi.org/10.1145/1111320.1111043

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

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

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

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

Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, London, UK, UK, 52–71. http://dl.acm.org/citation.cfm?id=648063.747438

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

Traian Florin Şerbanuţă and Grigore Roşu. 2010. 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)*. Springer, 104–122. https://doi.org/10.1007/978-3-642-16310-4_8

Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. 2014. 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) (LNCS)*, Vol. 8560. Springer, 425–440. https://doi.org/10.1007/978-3-319-08918-8_29

Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-Based Program Verifiers for All Languages. In *Proceedings of the 31th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 74–91. https://doi.org/10.1145/2983990.2984027

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

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

Chucky Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. 2009. A Rewriting Logic Approach to Type Inference. In *Recent Trends in Algebraic Development Techniques (Lecture Notes in Computer Science)*, Vol. 5486. Springer, 135–151. https://doi.org/doi:10.1007/978-3-642-03429-9 Revised Selected Papers from the 19th International Workshop on Algebraic Development Techniques (WADT'08).

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

Matthias Felleisen and D Friedman. 1986. Control Operators, the SECD Machine, and the λ-Calculus, Formal Description of Programming Concepts III (ed. M. Wirsing), 193–217. (1986).

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

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

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

Mark Hills, Feng Chen, and Grigore Roşu. 2008. 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) (ENTCS)*, Vol. To Appear. Elsevier.

Mark Hills and Grigore Roşu. 2008. Towards a Module System for K. In *Recent Trends in Algebraic Development Techniques (WADT'08) (Lecture Notes in Computer Science)*, Vol. 5486. 187–205.

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

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

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

P. Lincoln M. Clavel, S. Eker and J. Meseguer. 2000. Principles of Maude. In *Electronic Notes in Theoretical Computer Science*, J. Meseguer (Ed.), Vol. 4. Elsevier Science Publishers.

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

José Meseguer and Grigore Roşu. 2011. The Rewriting Logic Semantics Project: A Progress Report. In *Proceedings of the 17th International Symposium on Fundamentals of Computation Theory (FCT'11) (Lecture Notes in Computer Science)*, Vol. 6914. Springer, 1–37. https://doi.org/citation.cfm?id=2034214.2034215 Invited talk.

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

Brandon Moore and Grigore Roşu. 2015. *Program Verification by Coinduction.* Technical Report http://hdl.handle.net/2142/73177. University of Illinois.

Peter D Mosses. 2004. Modular structural operational semantics. *The Journal of Logic and Algebraic Programming* 60 (2004), 195 – 228. https://doi.org/10.1016/j.jlap.2004.03.008 Structural Operational Semantics.

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

Michael Norrish. 1998. *C formalised in HOL.* Technical Report.

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

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

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

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

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

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

Traian Florin Şerbănuţă, Gheorghe Ştefănescu, and Grigore Roşu. 2009. Defining and Executing P Systems with Structured Data in K. In *Workshop on Membrane Computing (WMC'08) (Lecture Notes in Computer Science)*, David W. Corne, Pierluigi Frisco, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa (Eds.), Vol. 5391. Springer, 374–393. https://doi.org/doi:10.1007/978-3-540-95885-7_26

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

Peter Sewell, Francesco zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok StrniŠa. 2010. Ott: Effective Tool Support for the Working Semanticist. *J. Funct. Program.* 20, 1 (Jan. 2010), 71–122. https://doi.org/10.1017/S0956796809990293

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

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