Universitat Politècnica de València

Departamento de Sistemas Informáticos y Computación

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Tesis de Máster

Inferencia de especificaciones para programas C

Candidato:
Daniel Pardo Pont

Directores:
María Alpuente Frasnedo
Alicia Villanueva García

Año Académico 2014-2015

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera, s/n
46022 Valencia
España

Resumen

A pesar de los abundantes beneficios que ofrecen, las especificaciones formales no acostumbran a emplearse en el desarrollo industrial de software. Con la finalidad de reducir el tiempo y el esfuerzo requerido para escribir especificaciones formales, en la presente Tesis de Máster se propone una técnica que obtiene de manera automática especificaciones a partir de código real. La metodología propuesta se basa en explotar las capacidades de ejecución simbólica recientemente proporcionadas por el marco $\mathbb{K}$ para inferir automáticamente especificaciones formales de programas escritos en un fragmento no trivial de C, denominado KernelC. En términos generales, el análisis simbólico de programas KernelC explica la ejecución de una función (modificadora) a través de otras rutinas (observadoras) del programa. Esta técnica ha sido implementada en la herramienta automática KindSpec 2.0, la cual genera axiomas que describen el comportamiento de entrada/salida de rutinas C que gestionan estructuras basadas en punteros (es decir, valores resultado y cambios de estado). En esta disertación se describe la implementación de dicho sistema y se analizan las diferencias respecto de trabajos previos relacionados con la inferencia de especificaciones sobre código C.

Palabras clave: especificaciones, inferencia automática, ejecución simbólica, semántica formal.

Resum

Tot i que ofereixen abundants beneficis, les especificacions formals no acostumen a emprar-se en el desenvolupament industrial de programari. Amb la finalitat de reduir el temps i l'esforç requerits per escriure especificacions formals, en la present Tesi de Màster es proposa una tècnica que obté de manera automàtica especificacions a partir de codi real. La metodologia proposta es basa en explotar les capacitats d'execució simbòlica recentment proporcionades pel marc $\mathbb{K}$ per tal d'inferir automàticament especificacions formals de programes escrits en un fragment no trivial de C, denominat KernelC. En termes generals, l'anàlisi simbòlica de programes KernelC explica l'execució d'una funció (modificadora) a través d'altres rutines (observadores) del programa. Aquesta tècnica ha estat implementada en la ferramenta automàtica KindSpec 2.0, la qual genera axiomes que descriuen el comportament d'entrada/eixida de rutines C que gestionen estructures basades en punters (és a dir, valors resultat i canvis d'estat). En aquesta dissertació es descriu la implementació de dit sistema i s'analitzen les diferències respecte de treballs previs relacionats amb la inferència d'especificacions sobre codi C.

Paraules clau: especificacions, inferència automàtica, execució simbòlica, semàntica formal.

Abstract

Despite its many unquestionable benefits, formal specifications are not widely used in industrial software development. In order to reduce the time and effort required to write formal specifications, in this Master Thesis we propose a technique for automatically discovering specifications from real code. The proposed methodology relies on the symbolic execution capabilities recently provided by the $\mathbb{K}$ framework that we exploit to automatically infer formal specifications from programs that are written in a non-trivial fragment of C, called KernelC. Roughly speaking, our symbolic analysis of KernelC programs explains the execution of a (modifier) function by using other (observer) routines in the program. We implemented our technique in the automated tool KindSpec 2.0, which generates axioms that describe the precise input/output behavior of C routines that handle pointer-based structures (i.e., result values and state change). We describe the implementation of our system and discuss the differences w.r.t. previous work on inferring specifications from C code.

Keywords: specifications, automatic inference, symbolic execution, formal semantics.

Contents

1 Introduction **1**

1.1 Motivation and Background 2

1.2 Related Work 5

1.3 Objectives of the Work 6

1.4 Structure of the Dissertation 7

2 Preliminaries **9**

2.1 Rewriting Logic 10

2.2 Maude 11

2.3 The $\mathbb{K}$ Framework 15

3 The KERNELC **Language** **19**

3.1 Introduction 20

3.2 Running Example 22

3.3 Specification of KERNELC in $\mathbb{K}$ 26

4 Symbolic Execution **33**

4.1 Introduction 34

4.2 The symbolic machinery in $\mathbb{K}$ 36

5 Inference Process **39**

6 Implementation of the System **45**

7 Conclusions and Future Work **51**

Bibliography **55**

Appendix: Full KERNELC **Specification** **59**

List of Figures

3.1 Abstract syntax of KernelC. 21
3.2 KernelC implementation of a doubly-linked list. 24
3.3 Expected specification for the `append(list,d)` function call. 25
3.3 Extended KernelC grammar specified in $\mathbb{K}$. 28
3.4 Desugaring rules for extended KernelC. 29
3.5 $\mathbb{K}$ concrete configuration of the implementation of KernelC. 31

6.1 Structure of the specification inference system KindSpec 2.0. 47
6.2 Example of `krun` output for the symbolic execution of `append`. 49
6.3 Computed specification for the `append(list,d)` function call. 50

1

Introduction

1.1 Motivation and Background

Formal specifications can be used for various software engineering activities ranging from documenting software to automated debugging, verification, and test-case generation. Specifications can take the form of contracts, interfaces, summaries, assumptions, invariants, properties, component abstractions, process models, rules, graphs, automata, etc. Despite its many unquestionable benefits, formal specifications are not widely used in industrial software development mainly due to the required writing effort, complexity, and tool support. Specification inference can help to mitigate these problems and is also useful for legacy program understanding and malware deobfuscation [5].

This thesis describes our steps towards a specification inference system for heap-manipulating programs that are written in a non-trivial fragment of C called KernelC [26], which includes functions, structures, pointers, and I/O primitives. We rely on the (rewriting logic) semantic framework $\mathbb{K}$ [25], which facilitates the development of executable semantics of programming languages and also allows formal analysis tools for the defined languages to be derived with minimal effort.

A language definition in $\mathbb{K}$ essentially consists of three parts: the *BNF* language syntax (annotated with $\mathbb{K}$ specific attributes), the structure of program configurations, and the semantic rules. Similarly to the classic operational semantics, program configurations contain an encoding for the environment, the heap, stacks, etc. and are represented as algebraic datatypes in $\mathbb{K}$. Program configurations organize the state in units called *cells*, which are labeled and can be nested.

For example, following the $\mathbb{K}$ notation, the program configuration

$$\Big\langle \ \langle \mathrm{tv}(int, 0) \rangle_{\mathsf{k}} \langle \mathtt{x} \mapsto \mathsf{x} \rangle_{\mathsf{env}} \langle \mathsf{x} \mapsto \ \mathrm{tv}(int, 5) \rangle_{\mathsf{heap}} \ \Big\rangle_{\mathsf{cfg}} \tag{1.1}$$

models the final state of a computation whose return value is the integer 0 (stored in the k cell, which contains the current code to be run), while program variable `x` (stored in the env cell) has the value 5 (stored in the memory address given by x in the heap cell, where information about pointers and data structures is recorded). Variables representing symbolic memory addresses are written in sans-serif font.

In $\mathbb{K}$, the configuration (1.1) is a friendly representation for the term

```
<cfg>
      <k> tv(int,0) </k>
      <env> x => pointer(x) </env>
```

```
    <heap> pointer(x) => tv(int,5) </heap>
</cfg>
```

Symbolic execution (SE) is a well-known program analysis technique that allows the program to be executed using *symbolic* input values instead of actual (concrete) data so that it executes the program by manipulating program expressions involving the symbolic values [20, 24]. Unlike concrete execution, where the path taken is determined by the input, in symbolic execution the program can take any feasible path. That path is given by a logical constraint on past and present values of the variables, called *path condition* because it is formed by constraints that are accumulated on the path taken by the execution to reach the current program point. Each symbolic execution path stands for many actual program runs (in fact, for exactly the set of runs whose concrete values satisfy the logical constraints). One of the traditional drawbacks of SE-based techniques is the high cost of decision procedures to solve path conditions. Recently, SE has found renewed interest due in part to the huge recent advances in decision procedures for logical satisfiability.

$\mathbb{K}$ semantics is traditionally compiled into Maude [8] for execution, debugging, and model checking. $\mathbb{K}$ implements reachability logic in the same way that Maude implements rewriting logic. In reachability logic, a particular class of first-order formulas with equality (encoded as (boolean) terms with logical variables and constraints over them) is used. These formulas, called *patterns*, specify those concrete configurations that match the pattern algebraic structure and satisfy its constraints. Since patterns allow logical variables and constraints over them, by using patterns, $\mathbb{K}$ rewriting becomes *symbolic execution* with the semantic rules of the language [3]. The SMT solver Z3 [10] is used in $\mathbb{K}$ for checking the satisfiability of the path constraints.

Symbolic execution in $\mathbb{K}$ relies on an automated transformation of both $\mathbb{K}$ configurations and $\mathbb{K}$ rules into corresponding symbolic $\mathbb{K}$ configurations (i.e., patterns) and symbolic $\mathbb{K}$ rules that capture all required symbolic ingredients: symbolic values for data structure fields and program variables; path conditions that constrain the variables in cells; multiple branches when a condition is reached during execution, etc. The transformed, symbolic rules define how symbolic configurations are rewritten during computation. Roughly speaking, each data structure field and program variable originally holds an initial, symbolic value. Then, by symbolically executing a program statement, the configuration cells (such as k, env and heap in the example above) are updated by mapping fields and variables to new symbolic values that

are represented as symbolic expressions, while the path conditions (stored in the path-condition cell) are correspondingly updated at each branching point.

For instance, the following pattern

$$\left\langle \begin{array}{c} \langle \mathrm{tv}(int, 0) \rangle_{\mathsf{k}} \\ \langle \cdots\ \mathtt{x} \mapsto \mathsf{x}, \mathtt{s} \mapsto \mathsf{s}\ \cdots \rangle_{\mathsf{env}} \\ \langle \cdots\ \mathsf{s} \mapsto (\mathtt{size} \mapsto \mathsf{?s.size}, \mathtt{capacity} \mapsto \mathsf{?s.capacity})\ \cdots \rangle_{\mathsf{heap}} \end{array} \right\rangle_{\mathsf{cfg}} \left\langle\ \mathsf{s} \neq \mathsf{NULL} \land \mathsf{?s.size} > 0\ \right\rangle_{\mathsf{path\text{-}condition}}$$

specifies the set of configurations as follows: (1) the k cell contains the integer value 0; (2) in the env cell, program variable `x` (in typographic font) is associated to the memory address x and `s` is bound to the pointer s; and (3) in the heap cell, the field size of s contains the symbolic value ?s.size (following the standard notation, symbolic values are preceded by a question mark). Additionally, `s` is not null and the value of its size field is greater than 0.

In this work, we redesign the technique of [1] for discovering specifications for heap-manipulating programs by adapting the symbolic infrastructure of $\mathbb{K}$ to support the specification inference process for KernelC programs. Specification inference is the task of discovering high-level specifications that closely describe the program behavior. Given a program `P`, the specification discovery problem for `P` is typically described as the problem of inferring a likely specification for every function `m` in `P` that modifies the state of encapsulated, dynamic data structures defined in the program. Following the standard terminology, any such function `m` is called a *modifier*. The intended specification for `m` is to be cleanly expressed by using any combination of the non-modifier functions of `P` (i.e., functions, called *observers*), which inspect the program state and return values expressing some information about the encapsulated data. However, because the C language does not enforce data encapsulation, we cannot assume purity of any function: every function in the program can potentially change the execution state, including the heap component of the state. In other words, any function can potentially be a *modifier*; hence we simply define an *observer* as any function whose return type is different from `void` (i.e., potentially expresses a property concerning the final *heap* contents or the return value of the function call).

The key idea behind our inference methodology was originally described in [1]. Given a *modifier* procedure for which we desire to obtain a specification, we start from an initial symbolic state s and symbolically evaluate m on s to obtain as a result a set of pairs (s, s') of initial and final symbolic states, respectively. Then, the observer methods in the program are used to explain the computed final symbolic states. This is achieved by analyzing the results of the symbolic execution of each

observer method o when it is fed with (suitable information that is easily extracted from) s and s'. More precisely, for each pair (s, s') of initial and final states, a pre/post statement is synthesized where the precondition is expressed in terms of the observers that *explain* the initial state s, whereas the postcondition contains the observers that *explain* the final state s'. To express a (partial) observational abstraction or explanation for (the constraints in) a given state in terms of the observer o, our criterion is that o computes the same symbolic values at the end of all its symbolic execution branches.

In contrast to [1], in this work we rely on the newly defined symbolic machinery for $\mathbb{K}$, while [1] was built on a symbolic infrastructure for KernelC that was manually developed in a quite ad-hoc and error prone way, by reusing some spare features of the formal verifier MatchC [28]. This strategic technological change will allow us to define a generic and more robust framework for the inference of specifications of languages defined within the $\mathbb{K}$ framework.

1.2 Related Work

The wide interest in program specifications as helpers for other analysis, validation, and verification processes have resulted in numerous approaches for (semi-)automatic computation of different kinds of specifications. Specifications can be property oriented (i.e., described by pre-/post conditions or functional code); stateful (i.e., described by some form of state machine); or intensional (i.e., described by axioms). In this work we focus on input-output relations: given a precondition for the state, we infer which modifications in the state are implied, and we express the relations as logical implications that reuse the program functions themselves, thus improving comprehension since the user is acquainted with them. A thorough comparison with the related literature can be found in [1]. Here we only try to cover those lines of research that have influenced our work the most.

Our axiomatic representation is inspired by [30], which relies on a model checker for symbolic execution and generates either Spec♯ specifications or parameterized unit tests. In contrast to [30], we take advantage of $\mathbb{K}$ symbolic capabilities to generate simpler and more accurate formulas that avoid reasoning with the global heap because the different pieces of the heap that are reachable from the function argument addresses are kept separate. Unlike our symbolic approach, Daikon [13] and DIDUCE [17] detect program invariants by extensive testing. Also, Henkel and Diwan [18] dynamically discover specifications for interfaces of Java classes by gener-

alizing the results of automated tests runs as an algebraic specification. QuickSpec [6] relies on the automated testing tool QuickCheck to distill general laws that a Haskell program satisfies. Whereas Daikon discovers invariants that hold at existing program points, QuickSpec discovers equations between arbitrary terms that are constructed using an API, similarly to [18]. AbsSpec [4] is a semantic-based inference method that relies on abstract interpretation and generates laws for Curry programs in the style of QuickSpec. A different abstract interpretation approach to infer approximate specifications is [29]. A combination of symbolic execution with dynamic testing is used in Dysy [9]. An alternative approach to software specification discovery is based on inductive matching learning: rather than using test cases to validate a tentative specification, they are used as examples to *induce* the specification (e.g., [31, 16]). Finally, Ghezzi *et al.* [15] infer specifications for container-like classes and express them as finite state automata that are supplemented with graph transformation rules.

This work improves existing approaches in the literature in several ways. Thanks to the handling of Maude's (hence $\mathbb{K}$'s) equational attributes [8], algebraic laws such as associativity, commutativity, or identity are naturally supported in our approach, which 1) leads to simpler and more efficient specifications, and 2) makes it easy to reason about typed data structures such as lists (list concatenation is associative with identity element *nil*), multisets (bag insertion is associative-commutative with identity $\emptyset$), and sets (set insertion is associative-commutative-idempotent with identity $\emptyset$). Since our approach is generic and not tied to the $\mathbb{K}$ semantics specification of KernelC, we expect the methodology developed in this work to be easily extendable to other languages for which a $\mathbb{K}$ semantics is given.

1.3 Objectives of the Work

Within the background described above, we can establish the following objectives for our work:

- To migrate the specification discovery technique of [1] to the holistic framework of the latest $\mathbb{K}$ release, which is based on symbolic execution, whereas [1] relied on the MatchC verification infrastructure of the old $\mathbb{K}$ platform, which is currently unsupported.
- To adapt the symbolic mechanism of $\mathbb{K}$ to deal with KernelC, also adapting and implementing the lazy initialization technique for manipulating complex

KernelC input data.

- To implement the specification inference technique in the KindSpec 2.0 system, building it on the capabilities of the SMT solver Z3 [10] not to only prove the accumulated path constraints as in $\mathbb{K}$ but also to incrementally simplify them on the fly.
- To integrate and put into practice the theoretical and practical skills acquired during the Master program. In particular, the knowledge on automated analysis and verification, multi-paradigm programming, logic and algebraic fundamentals, formal computation models and semantics of programming languages.

1.4 Structure of the Dissertation

The rest of the chapters of this document are organized as follows: Chapter 2 summarizes the technical background of Rewriting Logic and $\mathbb{K}$ that is needed for this thesis. Chapter 3 formalizes the language KernelC that is considered for the automated specification inference and show how we adapted it for the objectives of the work. Chapter 4 presents the key concepts regarding symbolic execution and the symbolic machinery of the $\mathbb{K}$ framework. Chapter 5 describes our specification discovery algorithm, and Chapter 6 shows how it is implemented in the prototype system KindSpec 2.0. Finally, Chapter 7 presents the conclusions of the work and discusses future research directions.

2

Preliminaries

In this section, we provide the technological infrastructure that makes the basis of our specification inference system.

2.1 Rewriting Logic

Rewriting Logic is a computational logic and a powerful semantic framework that can be used for naturally specifying a wide range of systems and languages in various application fields; for instance, models of concurrency and parallelism, network protocols or distributed algorithms. It is also a flexible metalogical framework for representing and mechanizing different logics and inference systems [14]. Rewriting Logic has been efficiently implemented in several languages, such as ASF+SDF, CafeOBJ or Maude, which are widely used for formal specification, analysis and verification.

In the context of programming languages, a language definition can be specified by means of a rewriting logic theory, which consists of a set of uninterpreted operations equationally constrained along with a set of rewrite rules that define the transitions which represent the evolution of the system. Formally speaking, a rewriting logic theory is a triple $T = (\Sigma, E, R)$ where (Σ, E) is an equational theory with signature Σ and equation set E, and R is the set of rewrite rules [23]. In this way, T represents a concurrent system whose states are elements of the algebra specified by (Σ, E), and whose transitions are specified by the rules in R. The initial state is provided as an uninterpreted Σ-term, which will evolve accordingly through its corresponding transition system until a state is achieved that cannot be further rewritten. Note that the equations in E describe the non-concurrent features of T whereas the set R defines the concurrent features.

The rewrite rules are ordered pairs of terms (accepted in the signature of the language) represented in the form $l \to r$, where both terms may contain variables. These rules are executed by a rewrite engine following the *match-and-apply* principle of term rewriting [21]: whenever a term t is obtained which can be seen as an instance of the left-hand-side l of a rule, say with a substitution θ, it is replaced by the right-hand-side r of the rule, maintaining the assignments of variables to values in θ. In other words, given $l \to r \in R$, and a fragment t of the current state such that $t = \theta(l)$, then $t := \theta(r)$. It is important to note that the substitution θ must be propagated to the whole term that represents the current state in order to keep consistency.

The rules of rewriting logic theories describe which local transitions may occur

in a certain state of the specified system, and allow us to reason about which general concurrent transitions are possible in a system satisfying such a description [23]. Thus, computationally, each rewriting step can be seen as a parallel local transition in the concurrent system. However, that is not the only possible reading: we can adopt a logical viewpoint, interpreting the rewrite rules as metarules for correct deduction in a logical system. In this case, the meaning of the rules turns into that of inference rules, indicating that the term at the right-hand-side can be derived from the expression at the left-hand-side.

Unlike most other logics, Rewriting Logic is fully neutral about the structure and properties of the expressions. The symbols and logical connectives in the signature Σ and their structural properties in the set of equational axioms E are entirely user-definable, which provides some interesting properties like great flexibility and the capacity to naturally express many different types of concurrent systems and represent many other logics in a general way.

2.2 Maude

Maude [7] is a high performance multiparadigm language that provides a very efficient implementation of Rewriting Logic. It was developed in 1993 as part of an international initiative to design a common platform for investigation, teaching and application of declarative languages[1]. As such, Maude presents a wide range of applications, including formal specification and verification of concurrent systems, declarative programming or theorem proving, to cite a few. In particular, the most interesting feature for the objectives of our work is its capacity to support executable environments and formal analysis tools for programming languages, computation models and logics as a powerful metalanguage based on logical reflection.

A Maude specification is composed by different modules. There are essentially two kinds of modules: functional modules, which define equational theories, and system modules, which describe rewrite theories. Rewrite rules can only appear in system modules and are declared with the keywords `rl` or `crl` (in the case of conditional rules, i.e., rules that can only be applied when certain conditions hold in the current state), whereas equations can appear in both types of modules and are declared with the keywords `eq` or `ceq` (for conditional equations). They both characterize the behavior of the function symbols of the signature, called operators, but carrying different meaning for the specification: the rewrite rules represent the

[1] Further information and an extensive manual can be found at `http://maude.cs.uiuc.edu`.

concurrent dynamics of the specified system, while the equations describe structural properties over operators. Since the inherent behavior of equations is non-concurrent, they are commonly oriented to be used for equational simplification of terms, thus improving efficiency.

Additionally, the language supports the declaration of certain equational axioms by means of operational attributes, i.e., specified together with the operators that satisfy them by using keywords. For instance, some typical algebraic properties like associativity (`assoc`), commutativity (`comm`), identity (`id`) or idempotence (`idem`) can be represented using this notation. This allows Maude to deal with these structural properties efficiently in a built-in way, even avoiding termination problems when the axiom can be applied repeatedly (as in the case of commutativity).

Example 1 *Let us consider the specification of a simple system such as a vending machine which dispenses coffee and cakes. The machine accepts dollars or dollar quarters as input and returns a coffee or a cake depending on the customer's selection. A coffee costs a dollar, and a cake costs three quarters. For simplicity, we consider that both cakes and coffee can only be bought using dollars, and that the machine can change four quarters into a dollar to still allow quarters as input. Whenever buying a cake with a dollar, the machine gives both the item and a quarter in return. These requirements are modeled by the following Maude specification:*

```
fmod VENDING-MACHINE-SIGNATURE is
  sorts Coin Item State .
  subsorts Coin Item < State .

  op null : -> State .
  op _ _ : State State -> State [assoc comm id: null] .
  op $ : -> Coin .
  op q : -> Coin .
  op cf : -> Item .
  op ck : -> Item .
endfm

mod VENDING-MACHINE is
  including VENDING-MACHINE-SIGNATURE .
  var St : State .
```

```
  rl [add-d] St => St $ .         --- Input of 1 dollar
  rl [add-q] St => St q .         --- Input of 1 quarter
  rl [buy-cf] $ => cf .           --- 1 coffee for 1 dollar
  rl [buy-ck] $ => ck q .         --- 1 cake and 1 quarter for 1 dollar

  eq [change] q q q q = $ .       --- 4 quarters = 1 dollar
endm
```

The keywords `fmod` *and* `endfm` *enclose a functional module, which we named* `VENDING-MACHINE-SIGNATURE`*, and the keywords* `mod` *and* `endm` *delimitate a system module identified as* `VENDING-MACHINE`*. As its name suggests, our functional module defines the signature of our system; i.e., the accepted operators along with their equational attributes and their types, called sorts. In Maude, every function symbol must belong to a sort explicitly indicated, and can take one or more arguments that are also sorted. The sorts can be nested by the keyword* `subsort`*, forming a hierarchy. In the example, the sorts* `Coin` *and* `Item` *are both specializations of the sort* `State`*, and the second operator in the list gives* `State` *the meaning of a list of mixed coins and items where each element is separated by a blank. Note that this operator is qualified by the algebraic axioms of associativity, commutativity (preventing the execution environment to endlessly loop in list inversions) and establishing null as the identity element of the list. With this, the operator* `null` *becomes the empty list of states, and every list of 1 or more* `State` *elements will implicitly end with* `null`*.*

Meanwhile, the system module `VENDING-MACHINE` *defines the rules and equations that describe the behavior of the system. In order to be able to work with the operators previously declared, the* `VENDING-MACHINE-SIGNATURE` *module has to be imported first using the keyword including. The rules and equations in Maude can be labeled (though this is optional), and they can contain sorted variables. The variables can be declared and bound to their sorts on their own, using the keyword* `var`*, or in the left-hand-side of the rule where they appear. Note that, although the coin input and item purchase functionalities have been modeled through rewrite rules, the change of four quarters into a dollar is represented by an equation, since the correspondence between those two units is a structural property of the operators and thus does not trigger a state transition.*

The Maude system includes an interpreter and some formal tools that allow Maude specifications to be: 1) executed, taking a term provided by the user as

initial state; 2) model checked, either with the search command of the interpreter or with Maude's LTL model checker; and 3) formally analyzed. In this way, just by specifying the semantics of any programming language as a Maude program, we get for free an interpreter and formal analysis tools for that language. This scales up to real languages, such as Java, C, Verilog or, in the case of this work, KernelC. However, although Maude specifications are inherently concurrent (except in the particular case that the whole system behavior is modeled through equations), the interpreter is sequential, so the concurrent behavior is simulated by interleaving sequential rewriting steps. In practice, this means that non-determinism can arise from the rewrite rules of the specification whenever the current state can be rewritten by applying two or more of those rules, generating a search tree in a way similar to logic programming languages. That does not happen with equations, though, since they represent structural properties of functions and the non-concurrent behavior of the system, hence being determinist. The equation set in a Maude specification is expected to be confluent and terminating, but since there is no compiler that automatically ensures it (thus, it can only be proven by external checkers), if two different equations are applicable in a determined context and lead to independent execution paths, the behavior of the system can become unpredictable and some computations may be lost.

Example 2 *(Example 1 continued) Consider again our specification for the vending machine. If we observe the rules of the system, they are clearly non-confluent (there are two possible, non-convergent transitions when the current state is* `$`*) and non-terminating (the machine can always get more money as input). Hence, if we were to run a search for all possible action sequences starting from a given initial state, it will not end since the search space is infinite. To illustrate how the non-determinism and search works in Maude, let us replace the system module in the previous specification with the following version without the non-terminating* `add` *rules:*

```
mod VENDING-MACHINE-TERMINATING is
  including VENDING-MACHINE-SIGNATURE .

  rl [buy-cf] $ => cf .          --- 1 coffee for 1 dollar
  rl [buy-ck] $ => ck q .        --- 1 cake and 1 quarter for 1 dollar

  eq [change] q q q q = $ .      --- 4 quarters = 1 dollar
endm
```

Now, if we run the search command of the interpreter to search the possible outcomes of the state `q q q q` *(four dollar quarters), we get these results:*

```
Maude> search q q q q =>! X:State .
  search in VENDING-MACHINE-TERMINATING : q q q q =>! X:State .

  Solution 1 (state 2)
  X:State --> cf

  Solution 2 (state 3)
  X:State --> ck q

  No more solutions.
```

Note that the four quarters state was deterministically rewritten to a `$` *state by the equation* `change`*, and then two parallel paths arose since there are two rules applicable to the rewritten state. After changing the state in each path accordingly to the corresponding rule, the search ends since there are no more possible rewritings, and the final states* `cf` *and* `ck q` *are returned as solutions.*

2.3 The 𝕂 Framework

𝕂 [27] is a rewrite-based framework for engineering language semantics. Given a syntax and a semantics of a certain language, 𝕂 generates a parser, an interpreter, and formal analysis tools such as model checkers and deductive theorem provers at no additional cost. It also supports various backends, such as Maude and, experimentally, Coq [25]. In other words, language semantics defined in 𝕂 can be translated into Maude or Coq definitions; and there is also a Java backend under development, which is expected to be released with 𝕂 4.0 version. There are even non-executable backends that compile 𝕂 specifications to visual formats, like LaTeX or PDF, allowing for automated documentation and document exchange.

In this work, we have chosen to use the Maude backend available in version 3.4 of 𝕂, the latest one that is stable, since the syntax and semantics of the specification language of 𝕂 are really close to Maude's, which improves the understandability of the 𝕂 specification and makes compilation and execution more efficient. Moreover, Maude's backend in the 3.4 version automatically extends the compiled language

with symbolic execution features, simplifying our preparatory work and making us easier to focus on the specification inference process.

A programming language definition in 𝕂 mainly consists of three components: the syntax of the constructs of the language, declared via the conventional *BNF* notation, the semantic rules which give meaning to those constructs, and the program configuration scheme. To build those elements with independence of the backend that will be used to compile them, 𝕂 provides a language on its own, known as *KIL* (which stands for 𝕂 Intermediate Language). However, some keywords and attributes of *KIL* hold a correspondence with Maude's, so we can predict how the functions of the 𝕂 specification will behave and create complex specifications without extensive training.

The productions accepted in the grammar of the language are declared through the keyword `syntax`, followed by the name of a non-terminal symbol which represents a syntactic category. The term declared can be either a terminal function symbol or another syntactic category, allowing non-terminals to be nested in order to establish a hierarchy. Provided that 𝕂 makes no distinction between algebraic signatures and their context-free notation, syntactic categories correspond to sorts and productions to operations in the signature [26]. As in Maude, the operators can be annotated with attributes which specify properties that the language construct satisfies. For this thesis' work, the most important attributes are associativity (declared with the keywords `left` or `right`, depending on the orientation) and strictness.

A language construct is strict in a subset of its arguments when those arguments have to be evaluated before the whole term. For instance, the variable assignment in imperative languages is a typical case of strict operation, since the value to be assigned can be either stored in a variable or pending to be computed by a function; then, the variable name or the function call are processed first to get the actual value. The strictness is specified using the keyword `strict(a)`, where `a` is a set of positions of the arguments of the operator separated by commas; or simply `strict` when all the arguments are strict. No order is assumed to the evaluation of two or more arguments of a function through strictness; if the arguments of an expression must be evaluated in a given order, then we say that the operator is sequentially strict and it is specified with the keyword `seqstrict`.

Program configurations are represented in 𝕂 as potentially nested structures of labeled cells (or containers) that represent the program state. They include a computation stack or continuation (named k), environments (env, heap), and a call

stack (stack), among others. 𝕂 cells can be lists, maps, (multi)sets of computations, or a multiset of other cells. Computations carry "computational meaning" as special nested list structures that sequentialize computational tasks, such as fragments of a program.

Rules in 𝕂 state how configurations (terms) evolve throughout the computation. 𝕂 rules are contextual; they mention a configuration context to which they apply, together with local changes they make to that context. As in Maude, 𝕂 distinguishes between the rules that model a (concurrent) transition of the system state, and the rules that represent structural properties over terms. However, in contrast to Maude, they are both defined with the same keyword, `rule`. The difference between the equivalent to Maude's rules and the counterpart to Maude's equations is provided by means of qualifier attributes, specifically the attribute `structural`; i.e., rules specified with the attribute `structural` will be translated into an equation when compiling the specification in the Maude backend, thus giving it a deterministic meaning. Conditions can also be specified for the semantic rules of a language, triggering the rewriting of the terms in the configuration only when those conditions hold. Whenever a rule is restricted by a condition, the directive `requires` is used, followed by a boolean expression representing the requirements that have to be satisfied in order for the rule to apply. In this way, rules defined with conditions in a 𝕂 specification translate through the Maude backend into either conditional rules `crl` or conditional equations `ceq`, depending on the companion qualifiers.

Similarly to configurations, rules can also be graphically represented and are split in two levels. Changes in the current configuration (which is shown in the upper level) are explicitly represented by underlining the part of the configuration that changes. The new value that substitutes the one that changes is written below the underlined part. As an example, we show the KernelC rule for assigning a value V of type T to the variable X. This rule uses three cells: k, env, and heap. The env cell is a mapping of variable names to their memory positions, whereas the heap cell binds the active memory positions to the actual values. Meanwhile, the k cell represents a stack of computations waiting to be run, with the left-most (i.e., top) element of the stack being the next computation to be undertaken.

$$\langle \frac{X = \mathrm{tv}(T,V)}{\mathrm{tv}(T,V)} \cdots \rangle_{\mathsf{k}} \langle \cdots\ X \mapsto \mathsf{X} \cdots \rangle_{\mathsf{env}} \langle \cdots\ \mathsf{X} \mapsto \frac{_}{\mathrm{tv}(T,V)} \cdots \rangle_{\mathsf{heap}}$$

This rule states that, if the next pending computation (which may be a part of the evaluation of a bigger expression) consists of the assignment $X = \mathrm{tv}(T,V)$, then

we look for X in the environment ($X \mapsto _$) and we update the associated mapping in the memory with the new value V of type T ($\mathrm{tv}(T, V)$). The value $\mathrm{tv}(T, V)$ is kept at the top of the stack (it might be used in the evaluation of the bigger expression). The rest of the cell's content in the rule does not undergo any modification (this is represented by the ··· card). This example rule reveals a useful feature of 𝕂: «rules only need to mention the minimum part of the configuration that is relevant for their operation». That is, only the cells read or changed by the rule have to be specified, and, within a cell, it is possible to omit parts of it by simply writing "··· ". For example, the rule above emphasizes the interest in: the instruction $X = \mathrm{tv}(T, V)$ only at the beginning of the k cell, and the mapping from variable X to its memory pointer X at any position in the env cell. Except for the subterms that are explicitly identified, upon variable assignment everything is kept unchanged.

The (desugared) 𝕂 rule for KernelC variable assignment is

```
rule    <k> X = tv(T,V) => tv(T,V) ...</k>
        <env>... X |-> pointer(X) ...</env>
        <heap>... pointer(X) |-> (_ => tv(T,V)) ...</heap>
```

where the underscore stands for an anonymous variable. The ellipses are also part of the desugared 𝕂 syntax and are used to replace the unnecessary parts of the cells. Hence, also in the desugared rule, the developers typically only mention the information that is absolutely necessary in their rules.

The 𝕂 tool offers support for semantics-based execution and formal analysis of programs. It is mainly composed of a compiler, called `kompile`, which transforms *KIL* code to the language or visual style of the selected backend, and an interpreter, called `krun`, which runs programs in the compiled language specifications. We can think of 𝕂 specifications as transition system generators, where the states are defined by the configurations and the transitions are described by the rules of the specification. Given a program and a list of arguments explicitly declared in the language specification, the 𝕂 interpreter is capable of creating an initial state (configuration) and then generating the possible behaviors of the program by means of transitions, applying the language semantics. The 𝕂 tool offers the possibility to take just one path or explore all feasible paths through a space search in a way similar to the Maude interpreter, thus providing the interpreter with both execution and model checking capabilities.

3

The KernelC Language

In this chapter, we describe the language KernelC that we consider for specification inference, and we justify the reasons for this selection. We also describe a KernelC program used as leading example to test the inference method, along with the expected inferred specification. Finally, we explain how the language specification was implemented in $\mathbb{K}$ and adapted to make it more suitable for our needs.

3.1 Introduction

Since it was conceived in the 70s, C (together with its variants, like C$^\sharp$ or C++) has become the most widely used imperative programming language. Different systems and applications have been developed on the basis of the C platform, ranging from simple text processors to performance-critical applications such as operating systems (specially those of the UNIX family) or video games. The main reason for the success of C is its capability to directly access the information stored in memory through pointer arithmetics, casting, and explicit allocation and deallocation, which allows developers to optimize the usage of computer's resources and obtain highest performance, yet generally achieving this at the expense of safety.

Testing our inference technique on C programs would be an interesting option that would instantly provide an important set of potential real and complex applications as a consequence of the widespread use of the language. However, C is a rather complex platform to be considered for the first try-outs of a system that infers specifications automatically from the complete semantics of a language. Some of its features, like the various data types that the language can handle or the importation of external function libraries, can cause the complexity of the inference process to grow to the point of turning it too much cost-expensive and even unpredictable, since in most cases the code of the imported libraries is not available (e.g., binaries installed in the operating system or built-in packages). That is why, in this thesis, we have chosen to start with a simplified language derived from the semantics of C, called KernelC.

KernelC is defined in [26] as a formal definition of the executable semantics of a non-trivial fragment of C, which includes essential imperative statements (variable assignment and lookup, arithmetic and logic operations and flow control directives, just to mention a few) as well as function definition and calling, declaration of data structures, pointers and routines for allocating and deallocating dynamic memory. The amount of primitive data types supported by C for variable values gets reduced to just one in KernelC: signed mathematical integers, i.e., the type identified by

Abstract syntax:

$$
\begin{aligned}
Nat &::= \mathbb{N} \\
Int &::= \mathbb{Z} \\
Id &::= \text{identifiers (as character strings)} \\
K &::= \ Nat|Int|Id \\
&\quad \mid\ !K \mid K \mathbin{\&\&} K \mid K \mathbin{||} K \\
&\quad \mid K\ op\ K, \quad op \in \{+, -, *, /, <, \leq, >, \geq, ==, !=\} \\
&\quad \mid K = K \\
&\quad \mid K; K \\
&\quad \mid \{K\} \\
&\quad \mid \{\} \\
&\quad \mid \text{NULL} \\
&\quad \mid malloc(K) \\
&\quad \mid free(K) \\
&\quad \mid *K \\
&\quad \mid \text{if } (K)\ K \mid \text{if } (K)\ K \text{ else } K \\
&\quad \mid \text{while } (K)\ K
\end{aligned}
$$

Desugaring of non-core constructs:

$$
\begin{aligned}
\text{NULL} &= 0 \\
!K &= \text{if } (K)\ 0 \text{ else } 1 \\
K1 \mathbin{\&\&} K2 &= \text{if } (K1)\ K2 \text{ else } 0 \\
K1 \mathbin{||} K2 &= \text{if } (K1)\ 1 \text{ else } K2 \\
\text{if } (K1)\ K2 &= \text{if } (K1)\ K2 \text{ else } \{\}
\end{aligned}
$$

Figure 3.1: Abstract syntax of KernelC.

the keyword `int`. Nevertheless, integer-type pointers and structured objects with integer (or pointer) fields are also handled by the language. These restrictions endow KernelC with the simplicity needed for developing the specification inference prototype tool and evaluating our system with reasonable effort, yet preserving enough capabilities and expressive power to write rather useful code, as we will show in Section 3.2.

The abstract syntax of KernelC was also defined in [26], and a summary in *BNF* notation is given in Figure 3.1. As stated before, the only arithmetic data considered by the language are integers; however, natural numbers and identifiers are also supported to represent memory addresses and variable names, respectively.

The constructs are grouped inside a unique syntactic category called K, which stands for the minimal infrastructure to define terms that is available in the $\mathbb{K}$ framework, and thus they do not belong to any specific sort. Arithmetic and relational operators, simple flow control statements (`if-else` conditionals and `while` loops), sequences of expressions and grouping blocks are the productions composing the syntax of KernelC. We can also use logical operators (AND, OR, and NOT), conditional statements without an `else` clause and the `NULL` value, although these are considered syntactic sugar for specific combinations of core constructs, and so they have to be desugared first in order to be interpreted.

Note that there are no productions in the shown syntax that serve for calling functions or variable declarations, and not even for defining struct types; the reason is that this abstract syntax establishes the indispensable core KernelC features although they can be extended as far as it is needed in each implementation. In our case, not only functions and structures will be considered in addition to the basic syntax, but also pointers of the undefined type `void*` and other ingredients that we will tackle later in Section 3.3.

3.2 Running Example

Our inference technique relies on the classification scheme developed in [22] for data abstractions, where a function (method) may be either a *constructor*, a *modifier* or an *observer*. A constructor returns a new object[1] of the class from scratch (i.e., without taking the object as an input parameter). A modifier alters an existing class instance (i.e., it changes the state of one or more of the data attributes in the instance). An observer inspects the object and returns a value characterizing one or more of its state attributes. We do not assume the traditional premise of the original classification in [22] that states that observer functions do not cause side effects on the state. This is because we want to apply our technique to any program, which may be written by third-party software producers that may not follow the observer purity discipline.

Let us introduce the leading example that we use to describe the inference methodology developed in this work and to assess the practicality of the prototype tool that implements it: a KernelC implementation of an abstract datatype for representing doubly-linked lists. Since the whole example includes a total of 13

[1] In KernelC, we understand for *object* an instance of a data structure type, and a data structure type is what we call a *class*.

methods, due to space restrictions we have chosen to comment on just one modifier and five observer methods (of which 2 are both modifiers and observers).

Example 3 *In the* KernelC *program of Figure 3.2, we represent a doubly-linked list as a data structure (*`struct List`*) that contains some content (field* `data`*), a pointer to the previous element in the list (field* `prev`*), and another pointer to the succesive element in the list (field* `next`*).*

A call `append(list,d)` *to the* `append` *function proceeds as follows: first, a new node* `new_node` *is allocated in memory; it is filled with the value* `d` *and its* `next` *pointer is initialized to* `NULL` *since it will become the last item in the list. Next, the function checks that the provided list* `list` *is not* `NULL`*, in which case it binds the* `next` *pointer of the final element of the list to the newly created node, and the* `prev` *pointer of the new node to the final node of* `list`*, then returns the pointer to the whole resulting* `list`*. Otherwise, when the input list* `list` *is null, then the* `prev` *pointer of* `new_node` *is initialized to* `NULL` *and the resulting full-fledged list that consists of one single element is simply returned.*

The observer function `length` *traverses the list by visiting every node in order to count the number of elements in the list. The observer function* `head` *returns the data field of the first node of the list;* `last` *delivers the data field of the last node of the list, which is done by first invoking* `reverse(list)` *to compute a mirrored version of the parameter* `list` *and then accessing the* `data` *field of its first node. The function* `init(list)` *returns the same list after removing the last item of the list. Finally, the observer* `find` *looks for the provided* `d` *value in the list, and returns* 1 *(which stands for* true*) if the* `d` *value is found; otherwise, the value* 0 *(which stands for* false*) is returned.*

From the program code of Example 3, for each modifier function m, we aim to synthesize an axiomatic specification that consists of a set of implication formulas $t_1 \Rightarrow t_2$, where t_1 and t_2 are conjunctions of equations of the form $l = r$. The left-hand side l of each equation can be either

- a call to an observer function and then r represents the return value of that call;
- the keyword `ret`, and then r represents the value returned by the modifier function m being observed.

```
#include <stdlib.h>

struct List {
  void* data;
  struct List* next;
  struct List* prev;
};

struct List* append(struct List* list,
    void* d) {
  struct List* new_node;
  struct List* final;

  new_node = (struct List*) malloc(sizeof
      (struct List));
  new_node->data = d;
  new_node->next = NULL;

  if (list != NULL) {
    final = list;
    if (final != NULL) {
      while (final->next != NULL)
        final = final->next;
    }
    final->next = new_node;
    new_node->prev = final;

    return list;
  }
  else {
    new_node->prev = NULL;
    list = new_node;
    return list;
  }
}

int length(struct List* list) {
  int len;

  len = 0;
  while (list != NULL) {
    len = len + 1;
    list = list->next;
  }
  return len;
}

struct List* reverse(struct List* list) {
  struct List* final;

  final = NULL;
  while (list != NULL) {
    final = list;
    list = final->next;
    final->next = final->prev;
    final->prev = list;
  }
  return final;
}

void* head(struct List* list) {
  if (list != NULL) {
      while (list->prev != NULL)
          list = list->prev;
  }
  return list->data;
}

struct List* last(struct List* list) {
  struct List* reversed;

  reversed = reverse(list);
  return head(reversed);
}

int find(struct List* list, void* d) {
  int found;

  found = 0;
  while (list != NULL && !(found)) {
      if (list->data == d)
        found = 1;
      else
        list = list->next;
  }
  return found;
}

struct List* init(struct List* list) {
  struct List* aux;

  if (list != NULL) {
     if (list->next != NULL) {
      aux = list->next;
      while (aux->next->next != NULL)
          aux = aux->next;
      aux->next = NULL;
     }
     else
      list = NULL;
  return list;
}
```

Figure 3.2: KernelC implementation of a doubly-linked list.

Informally, the statements on the left-hand and right-hand sides of the symbol $\Rightarrow$ are respectively satisfied before and after the execution of a function call to m. We adopt the standard primed notation for representing variable values after the execution.

Example 4 *Consider again the program of Example 3. The specification for the (modifier) function* `append` *that inserts an element* `d` *at the end of the list* `list` *is shown in Figure 3.3. The specification consists of two implications stating the*

$$\begin{pmatrix} \texttt{length(list)} = 0 \wedge \\ \texttt{reverse(list)} = \texttt{NULL} \wedge \\ \texttt{find(list,d)} = 0 \wedge \\ \texttt{init(list)} = \texttt{NULL} \wedge \\ \texttt{last(list)} = \texttt{NULL} \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{length(list}') = 1 \wedge \\ \texttt{reverse(list}') = \texttt{list} \wedge \\ \texttt{find(list}',\texttt{d}) = 1 \wedge \\ \texttt{init(list}') = \texttt{NULL} \wedge \\ \texttt{last(list}') = \texttt{d} \wedge \\ \texttt{ret} = \texttt{list}' \end{pmatrix}$$

$$\begin{pmatrix} \texttt{length(list)} = x \wedge \\ \texttt{length(list)} > 0 \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{length(list}') = x + 1 \wedge \\ \texttt{find(list}',\texttt{d}) = 1 \wedge \\ \texttt{last(list}') = \texttt{d} \wedge \\ \texttt{ret} = \texttt{list}' \end{pmatrix}$$

Figure 3.3: Expected specification for the `append(list,d)` function call.

conditions that are satisfied before and after the execution of a symbolic function call `append(list,d)`*. The first formula can be read as follows: if, before executing* `append(list,d)`*, the result of running* `length(list)` *is equal to* 0*, a call to* `find(list,d)` *returns* 0 *(since no value can be found in an empty list) and the results of executing* `reverse(list)`*,* `init(list)`*, and* `last(list)` *are all* `NULL` *(i.e., the list is empty), then, after executing* `append(list,d)`*, the length of the augmented list is* 1*, the reversed list coincides with the list itself, the value* `d` *can now be found in the list, the init segment of the list is* `NULL`*, the last element is the inserted value and the call returns the pointer to the (augmented) list. The second formula represents the general case: given a* `list` *with an arbitrary size* x*, the call* `append(list,d)` *causes the length to be increased by* 1*, the inserted value is found in the list, in particular it is returned by the* `last` *observer, and the (augmented) list is returned. Since the* `append` *function does not restrict the insertion to the cases in which the* `d` *value is still not inside the list, we cannot assume* `find` *to return* 0 *before running the modifier function* `append`*.*

Note that any implication formula in the specification may contain multiple facts (in

the pre- or post-condition) that refer to function calls that are assumed to be run independently under the same initial conditions. This avoids making any assumptions about function purity or side-effects.

3.3 Specification of KernelC in $\mathbb{K}$

A specification of the concrete syntax and formal semantics of KernelC was made by the $\mathbb{K}$ team and is available within the provided examples in the $\mathbb{K}$ tool distribution, yet it needs to be adapted to make it compatible with the 3.4 version of $\mathbb{K}$ (i.e., the only one currently providing symbolic execution features). Thus, in this work we use a modified version of the $\mathbb{K}$ specification for KernelC. In addition, we have added some capabilities, thus we had to adjust the configuration scheme and semantic rules in order to obtain an environment that fits best to our purposes.

A summary of the extended KernelC syntax of our approach is presented in Figure 3.3. In addition to the general definition that we saw in Figure 3.1, our specification considers the constructs from C that allow programmers to:

- handle typed values by means of the expression `tv(Type, Value)`,
- declare function profiles and structured data types,
- define functions, call them from other parts of the code and return values,
- declare local variables of types either `int`, `void`, structured types or pointers,
- access struct object fields (yet only through pointers using the operator `->`).

Some built-in elements of C are also taken in consideration: the headings `#include <stdio.h>` and `#include <stdlib.h>` are in this syntax since they are common imports used in real C programs, but no meaning is given to them so they will just be ignored. However, the functions `malloc`, `free` and `sizeof` from the library `stdlib` are included in the syntax, since they are relevant for the memory allocation and deallocation features of KernelC. Note that, unlike `sizeof`, the `malloc` and `free` functions are not considered as special constructs of the language but just particular cases of function calling for which the specification provides a specific meaning.

The language is also provided with other auxiliary sugared expressions, such as increment and decrement of variables (operators `++` and `--`), combinations of arithmetic operation and assignment (`+=`, `-=`, `*=` and `/=`), and return statements with no value associated, among others. However, in contrast to the abstract syntax

```
SYNTAX  File ::= Globals

SYNTAX  Globals ::= List{Global,“”}

SYNTAX  Global ::= StructDeclaration
                 | FunctionDeclaration
                 | FunctionDefinition
                 | #include <stdio.h>
                 | #include <stdlib.h>

SYNTAX  FunctionDeclaration ::= Type  Id(ParameterDeclarations) ;

SYNTAX  FunctionDefinition ::= Type  Id(ParameterDeclarations)StatementBlock

SYNTAX  ParameterDeclarations ::= List{ParameterDeclaration,“,”}

SYNTAX  ParameterDeclaration ::= Type  Id

SYNTAX  StructDeclaration ::= struct Id{VariableDeclarations} ;

SYNTAX  VariableDeclarations ::= List{VariableDeclaration,“”}

SYNTAX  VariableDeclaration ::= Type  Id ;

SYNTAX  PrimitiveType ::= int
                        | void
                        | Type *

SYNTAX  Type ::= PrimitiveType
               | struct Id

SYNTAX  FunctionProfile ::= no function
                          | Id(ParameterDeclarations)
                          | Id()

SYNTAX  StatementBlock ::= {VariableDeclarations  Statements}

SYNTAX  Statements ::= List{Statement,“”}

SYNTAX  Statement ::= Expression = Expression ; [strict(2)]
                    | Expression += Expression ; [strict(2)]
                    | Expression -= Expression ; [strict(2)]
                    | Expression *= Expression ; [strict(2)]
                    | Expression /= Expression ; [strict(2)]
                    | Expression ++ ;
                    | Expression -- ;
                    | Expression ; [strict]
                    | if (Expression)Statement else Statement [avoid]
                    | if (Expression)Statement
                    | while (Expression)Statement
                    | return Expression ; [strict(1)]
                    | return ;
                    | ;
                    | StatementBlock
```

```
SYNTAX  Expression ::= Int
                     | Id
                     | NULL
                     | (Expression) [bracket]
                     | Expression -> Id
                     | Id(Arguments) [strict(2)]
                     | sizeof (Type)
                     | (Type)Expression [strict(2)]
                     | - Expression [strict]
                     | * Expression [strict]
                     | & Expression
                     | Expression * Expression [strict]
                     | Expression / Expression [strict]
                     | Expression + Expression [strict]
                     | Expression - Expression [strict]
                     | Expression < Expression [seqstrict]
                     | Expression <= Expression [seqstrict]
                     | Expression > Expression [seqstrict]
                     | Expression >= Expression [seqstrict]
                     | Expression == Expression [strict]
                     | Expression != Expression [strict]
                     | ! Expression
                     | Expression && Expression [strict(1)]
                     | Expression || Expression [strict(1)]

SYNTAX  Expression ::= EvaluatedExpression

SYNTAX  EvaluatedExpression ::= TypedValue
                              | Bool
                              | String

SYNTAX  TypedValue ::= tv (Type, Value)

SYNTAX  Value ::= Int
                | Pointer
                | undef
                | objectValues (Map)

SYNTAX  Pointer ::= pointer (Expression)
                  | member (Expression, Id)
                  | null

SYNTAX  Arguments ::= List{Expression,“,”} [strict]

SYNTAX  EvaluatedArguments ::= List{EvaluatedExpression,“,”}

SYNTAX  Id ::= main
             | malloc
             | free
```

Figure 3.3: Extended KernelC grammar specified in $\mathbb{K}$.

RULE
$$\frac{I: Int}{\texttt{tv}\ (\ \texttt{int}, I)}$$
[structural]

RULE
$$\frac{\texttt{NULL}}{\texttt{tv}\ (\ \texttt{void *}, \texttt{null})}$$
[structural]

RULE
$$\frac{E1: Expression\ \texttt{+=}\ E2: Expression\ \texttt{;}}{E1\ \texttt{=}\ E1\ \texttt{+}\ E2\ \texttt{;}}$$
[structural]

RULE
$$\frac{E1: Expression\ \texttt{-=}\ E2: Expression\ \texttt{;}}{E1\ \texttt{=}\ E1\ \texttt{-}\ E2\ \texttt{;}}$$
[structural]

RULE
$$\frac{E1: Expression\ \texttt{*=}\ E2: Expression\ \texttt{;}}{E1\ \texttt{=}\ E1\ \texttt{*}\ E2\ \texttt{;}}$$
[structural]

RULE
$$\frac{E1: Expression\ \texttt{/=}\ E2: Expression\ \texttt{;}}{E1\ \texttt{=}\ E1\ \texttt{/}\ E2\ \texttt{;}}$$
[structural]

RULE
$$\frac{E: Expression\ \texttt{++}\ \ \texttt{;}}{E\ \texttt{=}\ E\ \texttt{+ 1 ;}}$$
[structural]

RULE
$$\frac{E: Expression\ --\ \ \texttt{;}}{E\ \texttt{=}\ E\ \texttt{- 1 ;}}$$
[structural]

RULE
$$\frac{\texttt{return ;}}{\texttt{return tv}\ (\ \texttt{void}, \texttt{undef})\ \texttt{;}}$$
[structural]

RULE
$$\frac{\texttt{if}\ (E: Expression) S: Statement}{\texttt{if}\ (E) S\ \texttt{else ;}}$$
[structural]

RULE
$$\frac{\texttt{while}\ (E: Expression) S: Statement}{\texttt{if}\ (E)\{S\ \texttt{while}\ (E) S\}}$$
[structural]

Figure 3.4: Desugaring rules for extended KernelC.

shown on Figure 3.1, the value `NULL` is not a syntactic sugar for the value 0 but for a typed value that represents initialized pointers with no associated memory. It must not be confused with uninitialized pointers or undefined values, for which the value `undef` exists. Furthermore, the logical operators AND, OR and NOT are not just derivative expressions of conditional clauses, but full-fledged syntactic expressions with a meaning on their own. The `while` loop is defined in terms of the conditional as usual. A list of the desugaring rules can be found in Figure 3.4.

Another important difference between the approach adopted for this thesis and the baseline defined in [26] lies in the interpretation of how KERNELC programs are structured. In our case, the files that contain KERNELC code do not simply consist of a sequence of instructions, but rather a set of struct and function definitions, with a layout close to that of real C code. That is why a non-terminal symbol named *File* has been syntactically declared as the top-level construct, and is composed by a list of *Globals*. Note that, without loss of generality, we assume KERNELC programs to not have global variables nor constants declared, thus simplifying the structure of code files.

The concrete configuration adopted in our KERNELC approach for computations and execution is outlined in Figure 3.5. In essence, it is composed by nine cells: the k cell is the standard 𝕂 cell that keeps track of the remaining computations to perform, and it is also known as the *continuation stack*. Since a program computation in 𝕂 is in practice a rewriting of the term at the top of the stack, the content of the k cell is not type-restricted but simply defined as a sequence of K elements. The cells env and heap represent the execution environment memory-wise: env is a mapping between variables to memory addresses in which their actual values are stored, whereas heap is the mapping between those addresses and its corresponding values. Next, struct and fun are repositories of structure data types and functions, respectively, which facilitates both the instantiation of structs in object variables and the change of context whenever a function is called. The cell locals is a list of the memory addresses that can be accessed in the current state, and so it is used to control forbidden memory accesses in terms of scope; e.g., if a variable is declared in a statement block and then referred to outside of the block. stack represents the call stack in the execution environment, allowing to restore the previous state when returning from a function call. Regarding function calls, the cell current-function-call, as its name suggests, holds the profile of the current function call (or either `no function` while initializing the environment), including the identifier of the method

$$\left\langle \begin{array}{c} \langle K \rangle_{\mathsf{k}}\ \langle Map \rangle_{\mathsf{env}}\ \langle Map \rangle_{\mathsf{heap}}\ \langle Map \rangle_{\mathsf{struct}}\ \langle Map \rangle_{\mathsf{fun}}\ \langle List \rangle_{\mathsf{locals}} \\ \langle List \rangle_{\mathsf{stack}}\ \langle FunctionProfile \rangle_{\mathsf{current-function-call}}\ \langle List \rangle_{\mathsf{scope}} \end{array} \right\rangle_{\mathsf{cfg}}$$

Figure 3.5: $\mathbb{K}$ concrete configuration of the implementation of KernelC.

and its arguments. Finally, the scope cell is a stack that stores the states (env and locals) to be restored whenever a change of scope happens.

The original $\mathbb{K}$ specification of KernelC was not structure-oriented, and so the struct cell did not exist. Neither did the cell scope, which was added as a part of an alternate method for restoring the state when changing the environment because the operations that were first used were incompatible with $\mathbb{K}$ 3.4. Regarding the remaining cells, they already existed in the original specification, but their content or use has been modified in order to orient the language toward the needs of the inference process. For instance, the cell current-function-call used to hold just the identifier of the function being executed, which did not allow to have overloaded methods, i.e., functions with the same name but different argument types. Furthermore, each frame of the call stack stack now stores different information, since the concept of *environment* has been augmented: some methods can return newly-created objects through pointers (what we defined as constructors before), so the local heap must not be erased when restoring the state before the call. And now there are more constructions available for the values associated to memory addresses in the heap, since we needed to add a representation for the struct objects. In this way, when compiling this language specification with the $\mathbb{K}$ tool, we will have not only a fully functional executable environment which can be used to run real programs with results similar to any other C interpreter or compiler, but also a powerful tool that simplifies the work of the specification inference system providing it with all the information it may need to carry out the inference process.

It is important to note that the language definition exposed in this section is just a fragment of a more structured and complex specification that also deals with internal aspects such as memory management, variable scoping or syntactic and runtime errors, to name a few. To get more detailed information about our KernelC approach's syntax and semantics, the whole specification is available for consulting in the Appendix. Also, there are more elements in both the syntax and the configuration of the language that have not been treated here since they are related to symbolic execution, but that will be explained in the next chapter.

4
Symbolic Execution

4.1 Introduction

Symbolic execution is a static analysis technique that consists of executing programs with symbolic values instead of concrete values. It proceeds like standard execution except that, when a function or routine is called, symbolic values are assigned to the actual parameters of the call and computed values become symbolic expressions that record all operations being applied. When symbolic execution reaches a conditional control flow statement, every possible execution path from this execution point must be explored. In order to keep track of the explored execution paths, symbolic execution also records the assumed (symbolic) conditions on the program inputs that determine each execution path in the so-called *path conditions* (one per possible branch), which are empty at the beginning of the execution. A path condition consists of the set of constraints that the arguments of a given function must satisfy in order for a concrete execution of the function to follow the considered path. Without loss of generality, we assume that the symbolically executed functions access no global variables; they could be easily modeled by passing them as additional function arguments.

Example 5 *Consider again the* `append` *function of Example 3. Assume that the input values for the actual parameters* `list` *and* `d` *are the symbolic pointer* list *and the symbolic value* `?d`*, respectively. Then, when the symbolic execution reaches the first* `if` *statement in the code, it explores the two paths arising from considering both the satisfaction and non-satisfaction of the guard in the conditional branching statement. The path condition of the first branch is updated with the constraint* $\mathsf{list} \neq \texttt{NULL}$*, whereas* $\mathsf{list} = \texttt{NULL}$ *is added to the path condition in the second branch.*

To summarize, symbolic execution can be represented as a tree-like structure where each branch corresponds to a possible execution path and has an associated path condition. The *successful* paths are those leading to a final (symbolic) configuration that encloses a satisfiable path constraint and that typically stores a (symbolic) computed result.

For the symbolic execution of KernelC programs, we must pay attention to pointer dereference and initialization. In C and, by extension, in KernelC, a structured datatype (`struct`) is an aggregate type that is used to comprise a nonempty set of sequentially allocated members, called fields, each of which has a name and a type. When a `struct` value is created, C uses the address of its first field to refer to the whole structure. In order to access a specific field `f` of the given structure type,

C computes `f`'s address by adding an offset (the sum of the sizes of each preceding field in the definition) to the address of the whole structure.

In our symbolic setting, the pointer arithmetics and memory layout machinery are abstracted by 1) using symbolic variables as addresses, instead of the conventional natural numbers; and 2) mapping each structure object into a single element of the heap cell that groups all object fields (and associated values). A specific field is then accessed by combining the identifiers of both the structure object and the field name.

Example 6 *Consider the structure type* `List` *of Example 3. The following configuration records a list variable* `l` *with: 1) the integer* 7 *in its* `data` *field; 2) a reference (pointer) named* pnode *as the value of its* `prev` *field; and 3) a reference (pointer)* nnode *as the value of its* `next` *field:*

$$\left\langle \ \ldots \langle \mathtt{l} \mapsto \mathsf{l} \rangle_{\mathsf{env}} \langle \cdots \ \mathsf{l} \mapsto (\mathtt{data} \mapsto \ \mathrm{tv}(\mathit{int}, 7), \mathtt{prev} \mapsto \mathsf{pnode}, \mathtt{next} \mapsto \mathsf{nnode}) \ \cdots \rangle_{\mathsf{heap}} \ldots \ \right\rangle_{\mathsf{cfg}}$$

In order to access a field of the list `l` *(e.g., its* `data` *field), the corresponding index is computed by juxtaposing the identifier of the* `data` *field to the pointer* l*, thus mimicking how the concrete access would be done in* C *(i.e.,* `l->data`*).*

Another critical point is the *undefinedness* problem that occurs in C programs when accessing uninitialized memory addresses. The KernelC semantics that we use preserves the concrete *well-definedness* behavior of pointer-based program functions of C while still detecting the *undefinedness* cases in a way similar to the C operational semantics of [12]. However, in our inference setting, we have no a priori information regarding the memory (specifically, information about the (un)initialized memory addresses). Therefore, when symbolic execution accesses (potentially uninitialized) memory positions, two cases must be considered: the case in which the memory is actually initialized and stores an object, and the case in which it stores a null pointer. In contrast to the approach described in [1], we do not consider cases where the pointer is undefined (i.e., when the execution is halted due to forbidden pointer access). This avoids accumulating too many solutions with undefined behavior, which could cause an explosion of axioms for programs that access new objects frequently, resulting in huge and redundant output specifications. For the case when the memory positions are actually initialized with non-null objects, a strategy to reconstruct the original object in memory is needed. We adapt the lazy initialization of objects of [19] to our setting: when a symbolic address (or address expression) is accessed for the first time, SE initializes the memory object that is located at

the given address with a new symbolic value. This means that the mapping in the heap cell is updated by assigning a new free variable to the symbolic address of the accessed field so that, from that point on, accesses to that field can only succeed. As a result, *undefined* computations can only occur in the case of syntactic program errors (i.e., expressions that are not accepted in the specification of the language).

Example 7 *(Example 5 continued) Before executing the first* `if` *statement for the first time, assume that the* heap *cell is empty, which means that nothing is known about the structure of the* heap*. After symbolically executing the guard of the* `while` *statement (which refers to the* `next` *field of the structured data* `final`*), by applying the lazy initialization approach, the* heap *cell gets updated to:*

$$\left\langle \begin{array}{c} \cdots \\ \langle \cdots\ \texttt{list} \mapsto \mathsf{list}, \texttt{final} \mapsto \mathsf{list}\ \cdots \rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto (\texttt{data} \mapsto \mathrm{undef}, \texttt{prev} \mapsto \mathrm{undef}, \texttt{next} \mapsto \mathsf{list.next}) \\ \mathsf{list.next} \mapsto \mathrm{undef} \end{array} \right\rangle_{\mathsf{heap}} \\ \cdots \end{array} \right\rangle_{\mathsf{cfg}}$$

In other words, new symbolic bindings for the actual parameters are added, which represent the assumptions we made over the corresponding data structures. More specifically, the accessed field is initialized with a fresh symbolic pointer list.next *whereas the fields that have not been accessed yet (temporarily) remain undefined, in a state that is specified by the symbolic constant* undef*.*

In the following section, we describe $\mathbb{K}$'s symbolic execution machinery and how we adapted it to support discovering program specifications.

4.2 The symbolic machinery in $\mathbb{K}$

Recently, the $\mathbb{K}$ framework has been enriched with a tool that automatically compiles language definitions into symbolic semantics. In other words, any language that is formally defined in $\mathbb{K}$ can (ideally) benefit, without cost, from symbolic execution. The $\mathbb{K}$ symbolic backend automatically attaches to the configuration a new cell, called path-condition, for the conditions on the input arguments that are accumulated during the symbolic execution. Roughly speaking, the mechanism works as follows: whenever a non-deterministic choice is found (i.e., the term at the top of the k cell can be rewritten by applying different rules), the symbolic engine considers each path independently, storing the assumptions that enable each concrete execution path in the path-condition cell. Therefore, the symbolic execution of programs under the $\mathbb{K}$

framework results in a set of *patterns* (consisting of the final symbolic configuration that encloses the corresponding path-condition cell) which we call *final patterns*.

Example 8 *Assume that our $\mathbb{K}$ specification contains these two rules, which represent the possible rewritings of an* `if` *statement:*

$$\langle \frac{\text{if } (\texttt{true})\ S \text{ else } _}{S} \cdots \rangle_{\mathsf{k}} \qquad \langle \frac{\text{if } (\texttt{false})\ _ \text{ else } S}{S} \cdots \rangle_{\mathsf{k}}$$

Now assume that we are running the following piece of code:

```
if (x > y) return 1; else return 0;
```

with symbolic variables `x` *and* `y`*, and no initial restrictions over them (i.e., the* path-condition *cell is initialized to* `true`*). The compilation of the language with $\mathbb{K}$'s symbolic backend explores both branches (i.e., the case when the guard* `x > y` *is true and the case when the guard evaluates to false), which respectively lead to the following patterns*[1]*:*

Branch 1: $\langle\ \langle \text{tv}(int, 1)\rangle_{\mathsf{k}} \ldots\ \rangle_{\mathsf{cfg}} \langle\ \mathsf{?x} > \mathsf{?y}\ \rangle_{\mathsf{path-condition}}$

Branch 2: $\langle\ \langle \text{tv}(int, 0)\rangle_{\mathsf{k}} \ldots\ \rangle_{\mathsf{cfg}} \langle\ \mathsf{?x} \leq \mathsf{?y}\ \rangle_{\mathsf{path-condition}}$

As already mentioned, the exhaustive symbolic execution of all paths cannot always be achieved in practice because an unbounded number of paths can arise in the presence of loops or recursion. We follow the standard approach to avoid the exponential blowup that is inherent in path enumeration by exploring loops up to a specified number of unfoldings. In our implementation, this is achieved by introducing extra conditions in the guards of the loops and additional parameters in the recursive functions where the limit of iterations or calls to be made is specified. This ensures that SE ends for all explored paths, thus delivering a finite (partial) represention of the program behavior [11]. Obviously, not all the potential execution paths are feasible, but $\mathbb{K}$ deals with this automatically and transparently to the user by using the theorem prover Z3 [10] to check the satisfiability of the path condition constraints.

It is important to note that the symbolic $\mathbb{K}$ engine is not endowed with the lazy initialization technique. As a consequence, any branching in $\mathbb{K}$'s symbolic execution trees is associated to the evaluation of a guarded instruction (conditional, while loop,

[1] We only write those cells that are relevant for the example.

etc.), whereas lazy initialization also adds bifurcations when mimicking the access to complex data structures (objects) because all possible scenarios are considered. In other words, branching is not only caused by the evaluation of guards (boolean expressions), but also by other kinds of expressions (for instance when assigning a value to a data structure).

Note that the path-condition cell (where constraints associated to guards are stored) is not under our control but is automatically handled by the $\mathbb{K}$ symbolic engine, which ensures language independence. For this reason, we have adopted the solution to introduce two new cells into the configuration, called init-struct and init-heap, that are used to store those constraints associated to non-guarded instructions that refer to complex data structures. These cells represent the initial memory for its corresponding symbolic execution branch, but in different formats and for different purposes: init-heap uses the same mapped notation as in heap, whereas init-struct represents the restrictions regarding the memory initialization through boolean expressions, more friendly and easy to join with the contents of the path-condition cell. Both of them are only modified when applying lazy initialization, hence it is natural that the memory stored in heap during computation does not satisfy some of the constraints of init-struct; for example, if the initial heap has a variable `list` initialized to NULL and then the program creates an actual object and stores it in `list`, init-struct will still contain the expression `list` = NULL, which does not hold anymore. By abuse, when we refer to the path condition ϕ of a pattern, we implicitly consider that ϕ includes the constraints in init-struct as well.

5

Inference Process

Let us introduce the basic notions that we use in our formalization. Given an input program, let $\mathcal{F}$ be the set of functions in the program. We distinguish the set of observers $\mathcal{O}$ and the set of modifiers $\mathcal{M}$. A function can be considered to be an *observer* if it explicitly returns a value, whereas any method can be considered to be a *modifier*. Thus, the set $\mathcal{O} \cap \mathcal{M}$ is generally non empty. For instance, the function `reverse` in Example 3 is both an observer and a modifier function.

Given a function $f \in \mathcal{F}$, we represent a call to f with the list of arguments *args* by $f(args)$. Then, $f(args)\{\phi\}$ is the $\mathbb{K}$ pattern built by inserting the call $f(args)$ at the top of the k cell and initializing the path condition cells with ϕ. This is helpful to start the execution of $f(args)$ under the (possibly non–empty) constraints of ϕ. We also denote by $\textsc{Se}(f(args)\{\phi\})$ the set of final patterns obtained from the symbolic execution of the pattern $f(args)\{\phi\}$ (i.e., the leaves of the deployed symbolic execution tree).

Our specification inference methodology is formalized in Algorithm 1. First, the

Algorithm 1 Specification Inference.

Require: $m \in \mathcal{M}$ of arity n;
1. $S = \textsc{Se}(m(\mathsf{a}_1, \ldots, \mathsf{a}_n)\{\texttt{true}\})$
2. $axiomSet := \emptyset$;
3. **for all** $p \in S$ with path-condition cell ϕ_p, init-heap cell φ and return value v **do**
4. $eqs_{pre} := explain(\langle\langle m(\mathsf{a}_1, \ldots, \mathsf{a}_n)\rangle_{\mathsf{k}} \langle\varphi\rangle_{\mathsf{heap}} \ldots\rangle_{\mathsf{cfg}} \langle\phi_p\rangle_{\mathsf{path-condition}}, [\mathsf{a}_1, \ldots, \mathsf{a}_n]);$
5. $eqs_{post} := explain(p, [\mathsf{a}_1, \ldots, \mathsf{a}_n]);$
6. $eq_{ret} := (ret = \mathsf{v});$
7. $axiomSet := axiomSet \cup \{eqs_{pre} \Rightarrow (eqs_{post} \cup eq_{ret})\};$
8. **end for**
9. spec := $simplify(axiomSet)$
10. **return** spec

modifier method of interest m is symbolically executed with fresh symbolic variables $\mathsf{a}_1, \ldots, \mathsf{a}_n$ as arguments and empty constraint `true`, and the set S of final patterns is retrieved from the leaves of the symbolic tree. For each pattern in S, the corresponding path condition is simplified (by calling the automated theorem prover Z3) to avoid redundancies and simplify the analysis. Then, we proceed to compute an axiom for each pattern p in S that explains (by using the observers) the properties that hold in the state before and after the execution of the method. This is done by means of the function $explain(q, as)$, where q is a pattern and as is a list of symbolic variables, given in Algorithm 2. The explanation for initial states whose symbolic execution end in p (line 4) must ensure that the input data comply with the conditions that make the path to p feasible, which is achieved by imposing the

conditions given by ϕ to the input pattern to be explained (i.e. by feeding its heap cell with φ before invoking the routine *explain*). Then, we proceed to explain (also with the observers) the properties of the considered final state (the final pattern p) by invoking $explain(p, [\mathsf{a}_1, \ldots, \mathsf{a}_n])$. Finally, the return value v is retrieved from the k cell of p, and the axiom $ret = \mathsf{v}$ is added to the specification inferred. This value could be either undefined or a single typed value that represents the return from the function m under the conditions given by ϕ.

The computed axioms are implications of the form $l_i \Rightarrow r_i$, where l_i is a conjunction of preconditions and r_i is a conjunction of postconditions. Note that a conjunction of equations is represented as an equation set in Algorithm 2. The function *simplify* implements a post-processing which consists of: (1) disjoining the preconditions l_i that have the same postcondition r_i and simplifying the resulting precondition; and (2) conjoining the postconditions r_i that share the same precondition and simplifying the resulting postcondition.

Let us illustrate the application of the inference algorithm with the following example.

Example 9 *Let us compute a specification for the* `append` modifier *function of Example 3. Following the algorithm, we first compute* SE(`append(list,d)`[`true`]) *with* `list` *and* `d` *(free) symbolic variables. Since there are no constraints in the initial symbolic configuration, the execution covers all possible initial concrete configurations. For simplicity, we set the number of loop unrollings to one; as a consequence, the symbolic execution computes three final patterns. The following pattern e represents the final state for the path where the body of the* `while` *statement never gets executed (0 iterations):*

$$\left\langle \begin{array}{c} \langle \mathrm{tv}(\mathit{struct\ List*}, \mathsf{list}) \rangle_{\mathsf{k}} \\ \langle \mathtt{list} \mapsto \mathsf{list}, \mathtt{d} \mapsto \mathsf{d}, \mathtt{new_node} \mapsto \mathsf{new_node}, \mathtt{final} \mapsto \mathsf{list} \rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto (\mathtt{data} \mapsto \mathrm{undef}, \mathtt{prev} \mapsto \mathrm{undef}, \mathtt{next} \mapsto \mathsf{new_node}) \\ \mathsf{new_node} \mapsto (\mathtt{data} \mapsto \mathsf{d}, \mathtt{prev} \mapsto \mathsf{list}, \mathtt{next} \mapsto \mathsf{NULL}) \\ \mathsf{d} \mapsto \mathrm{tv}(\mathit{void}, ?\mathsf{d}) \end{array} \right\rangle_{\mathsf{heap}} \\ \langle \mathsf{list} \neq \mathsf{NULL} \ \wedge \mathsf{list}\text{->}\mathtt{next} = \mathsf{NULL} \rangle_{\mathsf{init-struct}} \end{array} \right\rangle_{\mathsf{cfg}}$$

The execution of this path returns the pointer to the resulting list: the returned pointer is represented by the typed value tv(*struct List*∗, list) *in the* k *cell. The field* list->`next` *is accessed only after checking that* list *is not null: it has been assumed* `list != NULL` *at the first conditional expression, thus the constraint* list ≠ NULL *has been added to the path condition, whereas* `final->next != NULL` *(the guard of the*

`while` *loop) is assumed false, thus the constraint* list->next = NULL *has been gathered. Note that, although the variable accessed in the code is* `final`*, the generated path constraint refers to the pointer* list *since both* `final` *and* `list` *are bound to the same memory address in the environment.*

Let us now describe Algorithm 2 which defines the function $explain(q, as)$. Given a $\mathbb{K}$ pattern q and a list of symbolic variables as, this function describes q as a set of equations that are obtained by executing the observer functions in the state. Each equation relates the call to an observer function (or *built-in* function) with the (symbolic) value that the call returns. In the algorithm, $As \sqsubseteq as$ means that the list of elements As is a permutation of some (or all) elements in as.

Algorithm 2 Computing explanations: $explain(q, as)$

Require: q : the pattern to be explained (with path condition ϕ)
Require: as : a list of symbolic variables
1. $\mathcal{C}$: the universe of observer calls;
2. $eqSet := \emptyset$;
3. **for all** $o(As) \in \mathcal{C}$ with $As \sqsubseteq as$ **do**
4. $\quad S = \text{SE}(o(As)\{\phi\})$
5. $\quad$ **if** $\nexists\ q_1, q_2 \in S$ s.t. q_1 and q_2 contain a different return value k in their k cell **then**
6. $\quad\quad eqSet := eqSet \cup (o(As) = \mathsf{k})$
7. $\quad$ **end if**
8. **end for**
9. **return** $eqSet$

Roughly speaking, given a pattern q, $explain(q, as)$ first generates the universe of observer function calls $\mathcal{C}$, which consists of all the function calls $o(As)$ that satisfy that:

- o belongs to $\mathcal{O}$ or to the set of (predefined) built-in functions,
- the argument list $As \sqsubseteq as$ respects the type and arity of o.

Then, for each call $o(As) \in \mathcal{C}$, Algorithm 2 checks whether all the final symbolic configurations (leaves) resulting from the symbolic execution of $o(As)$, under the constraints given by ϕ, have the same return value. When the call satisfies this requirement, an equation is generated (line 6 in Algorithm 2). Otherwise, the observation is inconclusive and no explanation is delivered in terms of the executed observer function. The algorithm finally returns the set of all the explanatory equations inferred.

Example 10 (Example 9 continued) *Let us show how we compute the explanation for the final state of pattern p in Example 9. Given the observer functions* `length`*,* `reverse`*,* `head`*,* `last`*,* `find`*, and* `init`*, and the symbolic variables* `list` *and* `d`*, the universe of observer calls is* `length(list)`*,* `reverse(list)`*,* `head(list)`*,* `last(list)`*,* `find(list,d)`*, and* `init(list)`*. Let us consider the case for the observer call* `length(list)` *in detail.*

When we symbolically execute `length(list)` *on the pattern p, we obtain a single final pattern:*

$$\left\langle \begin{array}{c} \langle \mathrm{tv}(int, 2) \rangle_{\mathsf{k}} \\ \langle \mathtt{list} \mapsto \mathsf{list}, \mathtt{length} \mapsto \mathsf{length} \rangle_{\mathsf{env}} \\ \left\langle \begin{array}{c} \mathsf{list} \mapsto (\mathtt{data} \mapsto \mathrm{undef}, \mathtt{prev} \mapsto \mathrm{undef}, \mathtt{next} \mapsto \mathsf{new_node}) \\ \mathsf{new_node} \mapsto (\mathtt{data} \mapsto \mathsf{d}, \mathtt{prev} \mapsto \mathsf{list}, \mathtt{next} \mapsto \mathsf{NULL}) \\ \mathsf{d} \mapsto \mathrm{tv}(void, ?\mathsf{d}) \\ \mathsf{length} \mapsto \mathrm{tv}(int, 2) \end{array} \right\rangle_{\mathsf{heap}} \\ \langle \mathsf{list} \neq \mathsf{NULL} \wedge \mathsf{list}\text{->}\mathtt{next} = \mathsf{NULL} \rangle_{\mathsf{init-struct}} \end{array} \right\rangle_{\mathsf{cfg}}$$

Since there are no observer paths returning different values and the associated return value is the integer 2, then the equation $\mathtt{length}(\mathtt{list}) = 2$ *is computed as a (partial) explanation for the final pattern under consideration. Thus, this term is added to the set of equations eqSet that are computed by Algorithm 2.*

6
Implementation of the System

A prototype implementation of the specification inference methodology presented in this thesis has been developed in the automated tool KindSpec 2.0. In this implementation chapter, we explain the system's structure and operation and discuss the results obtained for the running example of Section 3.2.

Roughly speaking, our specification inference system consists of a Java application that takes as input: 1) a file that contains the source code and 2) the identifier of the function to be specified (which is assumed to be defined inside that file). The outcome is a list consisting of the axioms of the discovered formal specification. The hard work is carried out by a modular structure which is shown in Figure 6.1, and sequentially performs the computations needed to obtain the desired specification. The process is as follows:

1. The front-end module calls out the $\mathbb{K}$ interpreter, `krun`, by using the command line of the operating system. The arguments of the call are: the name of the input function, symbolic values for all of its arguments, and a non-restricted environment (empty initial heap and `true` path condition). The $\mathbb{K}$ tool must be installed in the computer in order to run the system, since it is an external component.

2. The $\mathbb{K}$ interpreter uses the compiled definition of KernelC to symbolically execute the selected function and obtain the possible solutions (each one representing a feasible path). The compiled language must be inside the application project (and binaries) due to a specific feature of the $\mathbb{K}$ tool: the extension of the file that contains the source code is not restricted (it could even be a plain text `.txt` file), but the language must be available in the same directory of the file system where the call is made. This does not prevent our system from being platform-independent; however, makes it more difficult to operate since the working directory would have to change depending on the language.

3. The main module (the one in charge of the inference process) reads the output of the interpreter by means of a buffer, splitting the information in solutions and cells. First, it explores the fun repository to identify the observer functions in the file; i.e., those whose return type is different from `void`. Then, for each solution obtained, the return value is stored (to add the equation eq_{ret} to the corresponding axiom, if any), and so are the heap and init-heap cells, which characterize the final and initial patterns to be explained, respectively.

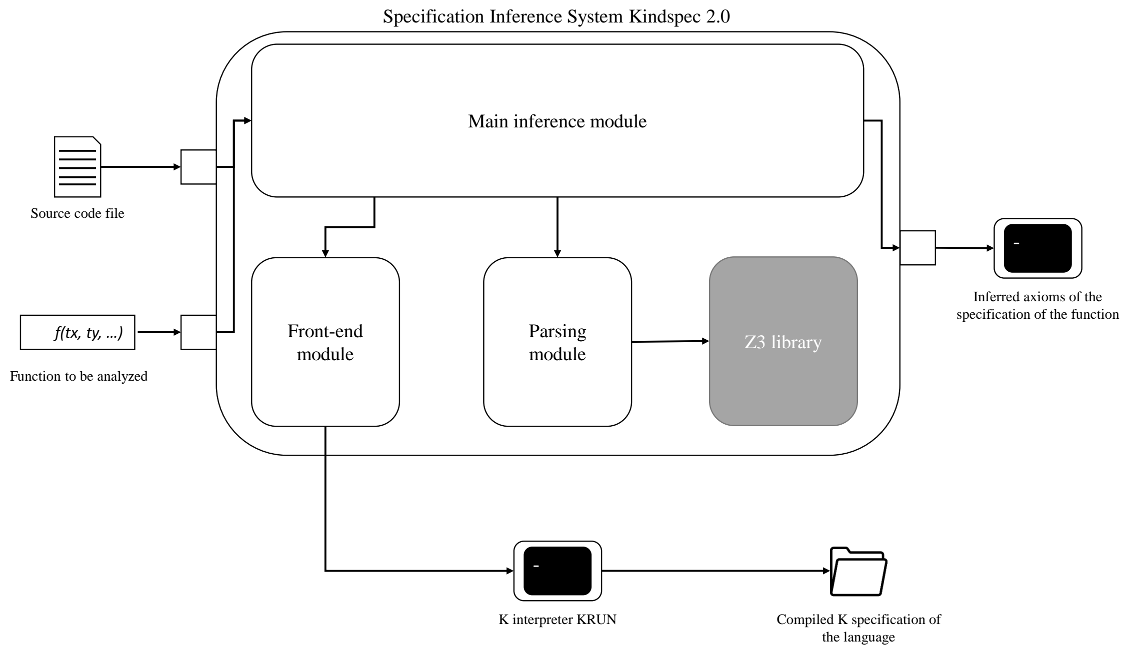


Figure 6.1: Structure of the specification inference system KindSpec 2.0.

4. The symbolic engine of $\mathbb{K}$ does actually accumulate the path constraints in bifurcation points (such as conditionals or loops) and prune unfeasible paths, but the expressions in those constraints are not reduced, which can cause the path conditions to contain junk and produce confusion. That is why, before storing them and use them to create the initial and final patterns of each solution, they are sent to the parsing module, which converts them into Z3 constructions in order to call the SMT solver for simplification. This module is built upon the tools JFlex[1] and CUP[2], which provide an abstraction to create scanners and parsers for custom languages in a rather simple way and offer the capability to generate the corresponding Java code and attach it to any project. After this simplification, the path conditions are translated back into $\mathbb{K}$ native language and stored.

5. Now that the main module has been provided with the necessary information, the inference algorithm formulated in Section 5 is applied. For each solution s obtained from the symbolic execution of the modifier, and each observer o recognized while scanning the fun cell (whose parameters are a subset of the profile of the modifier), o is executed under the environment of the initial and

[1] http://jflex.de/
[2] http://www2.cs.tum.edu/projects/cup/

final patterns of s to check whether all the possible paths return the same value (or if there is only one possible path). The comparison of the return values is done by means of string equality (the construct for values contains both the type and the value, so no more information is needed and the value does not need to be parsed). Again, this execution is materialized by the front-end module, who calls the $\mathbb{K}$ interpreter with the suitable arguments to set up the right context. To explain the initial pattern of a solution, the initial heap provided to the observer is the content of the initial-heap cell, whereas to explain the final pattern the data to be passed in is the content of the heap cell. The reason of this difference lies in the fact that, to run an observer function *after* the execution of the modifier, the final state of the memory must be preserved.

6. Finally, the axioms sets are built, simplified, and printed in the standard output (in our test run, the OS command terminal).

As we can observe, the operation of the system implies text processing to a great extent; after all, the way the front-end module and the $\mathbb{K}$ interpreter communicate is through a text-driven command console. That is one of the reasons why we chose Java as the platform to build our specification inference tool: the String type is much simpler to handle in Java than in C even providing methods to easily interpret the output from the command line and navigate through it. Another important reason is that, since the $\mathbb{K}$ tool is expected to be provided with a Java backend soon, it may be possible to attach language compilations directly into the project, hence simplifying the system so that its binaries do not have to worry anymore about directories and working environments.

To get a glimpse of the information that the KindSpec 2.0 system will process, Figure 6.2 showcases an example of how the output of `krun` looks like. The figure shows the information associated to the (symbolic) path that ends after in the body of the `else` clause (i.e., when the list passed as an argument is `NULL`) of the function `append` from Example 3. Note that the init-struct and init-heap cells are consistent with each other, whereas the (final) heap cell contents represent the memory after the execution, thus the information stored in it is different to that described in the restrictions of the init-struct cell. We also want to highlight how the storage of struct objects is implemented in our approach: we use an auxiliary constructor symbol named `objectValues` whose argument is a mapping from field identifiers to the values associated to them. Finally, note the presence of a new program variable `contador` that did not appear in Example 3, which is the auxiliary variable that we

KernelC — bash — 107×54

```
Solution 4:
<path-condition>
    true
</path-condition>
<Cfg>
    <k>
        tv ( struct List * , pointer ( (* new_node) ) )
    </k>
    <current-function-call>
        append ( struct List * list, void * d )
    </current-function-call>
    <init-struct>
        pointer ( d ) == null andBool pointer ( list ) == null
    </init-struct>
    <stack>
        .List
    </stack>
    <scope>
        .List
    </scope>
    <locals>
        ListItem(tv ( struct List * * , pointer ( list ) ))
        ListItem(tv ( void * * , pointer ( d ) ))
        ListItem(tv ( struct List * * , pointer ( new_node ) ))
        ListItem(tv ( struct List * * , pointer ( final ) ))
        ListItem(tv ( int * , pointer ( contador ) ))
        ListItem(tv ( struct List * , pointer ( (* new_node) ) ))
    </locals>
    <struct>
        List |-> structFields ( ListItem(field ( data , void * ))
          ListItem(field ( next , struct List * )) ListItem(field ( prev ,
          struct List * )) )
    </struct>
    <env>
        contador |-> tv ( int * , pointer ( contador ) )
        d |-> tv ( void * * , pointer ( d ) )
        final |-> tv ( struct List * * , pointer ( final ) )
        list |-> tv ( struct List * * , pointer ( list ) )
        new_node |-> tv ( struct List * * , pointer ( new_node ) )
    </env>
    <init-heap>
        pointer ( d ) |-> tv ( void * , null )
        pointer ( list ) |-> tv ( struct List * , null )
    </init-heap>
    <heap>
        pointer ( (* new_node) ) |-> tv ( struct List , objectValues (
          data |-> tv ( void * , null ) next |-> tv ( struct List * , null )
          prev |-> tv ( struct List * , null ) ) )
        pointer ( contador ) |-> tv ( int , 0 )
        pointer ( d ) |-> tv ( void * , null )
        pointer ( final ) |-> tv ( struct List * , undef )
        pointer ( list ) |-> tv ( struct List * , pointer ( (* new_node) ) )
        pointer ( new_node ) |-> tv ( struct List * , pointer ( (* new_node) )
          )
    </heap>
    <fun>
        append |-> functionBody ( struct List * list, void * d , struct
          List * , { { struct List * new_node ; struct List * final ; int
          contador ; contador = 0 ; new_node = (( struct List * ) (malloc ( (
          sizeof ( struct List )) ))) ; (new_node -> data) = d ; (new_node ->
          next) = (NULL) ; if ( (list != (NULL)) ) {  final = list ; if ( (
          final != (NULL)) ) {  while ( (((final -> next) != (NULL)) && (
          contador < 1)) ) {  final = (final -> next) ; contador ++ ; } } (
          final -> next) = new_node ; (new_node -> prev) = final ; return list
           ; } else {  (new_node -> prev) = (NULL) ; list = new_node ; return
          list ; } } return ; } )
        find |-> functionBody ( struct List * list, void * d , int , { { int
          found ; found = 0 ; while ( ((list != (NULL)) && (! found)) ) {  if
          ( ((list -> data) == d) ) found = 1 ; else list = (list -> next) ; }
           return found ; } return ; } )
        head |-> functionBody ( struct List * list , void * , { {  if ( (list
          != (NULL)) ) {  while ( ((list -> prev) != (NULL)) ) list = (list ->
           prev) ; } return (list -> data) ; } return ; } )
        init |-> functionBody ( struct List * list , struct List * , { {
          struct List * aux ; if ( (list != (NULL)) ) {  if ( ((list -> next)
          != (NULL)) ) {  aux = (list -> next) ; while ( (((aux -> next) ->
          next) != (NULL)) ) aux = (aux -> next) ; (aux -> next) = (NULL) ; }
          else list = (NULL) ; return list ; } } return ; } )
        last |-> functionBody ( struct List * list , struct List * , { {
          struct List * reversed ; reversed = (reverse ( list )) ; return (
          head ( reversed )) ; } return ; } )
        length |-> functionBody ( struct List * list , int , { { int len ; int
           contador ; len = 0 ; while ( (list != (NULL)) ) {  len = (len + 1)
          ; list = (list -> next) ; } return len ; } return ; } )
        reverse |-> functionBody ( struct List * list , struct List * , { {
          struct List * final ; final = (NULL) ; while ( (list != (NULL)) ) {
           final = list ; list = (final -> next) ; (final -> next) = (final ->
           prev) ; (final -> prev) = list ; } return final ; } return ; } )

    </fun>
</Cfg>
```

Figure 6.2: Example of `krun` output for the symbolic execution of `append`.

$$\begin{pmatrix} \texttt{length}(\texttt{list}) = 0 \wedge \\ \texttt{reverse}(\texttt{list}) = \texttt{NULL} \wedge \\ \texttt{find}(\texttt{list}, \texttt{data}) = 0 \wedge \\ \texttt{init}(\texttt{list}) = \texttt{NULL} \wedge \\ \texttt{last}(\texttt{list}) = \texttt{NULL} \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{length}(\texttt{list}') = 1 \wedge \\ \texttt{reverse}(\texttt{list}') = \texttt{list} \wedge \\ \texttt{find}(\texttt{list}', \texttt{data}) = 1 \wedge \\ \texttt{init}(\texttt{list}') = \texttt{NULL} \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{pmatrix}$$

$$\begin{pmatrix} \texttt{length}(\texttt{list}) = 1 \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{length}(\texttt{list}') = 2 \wedge \\ \texttt{find}(\texttt{list}', \texttt{data}) = 1 \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{pmatrix}$$

$$\begin{pmatrix} \texttt{length}(\texttt{list}) = 2 \end{pmatrix} \Rightarrow \begin{pmatrix} \texttt{length}(\texttt{list}') = 3 \wedge \\ \texttt{find}(\texttt{list}', \texttt{data}) = 1 \wedge \\ \texttt{last}(\texttt{list}') = \texttt{data} \wedge \\ \texttt{ret} = \texttt{list}' \end{pmatrix}$$

Figure 6.3: Computed specification for the `append(list,d)` function call.

used to keep track of the number of loop unrollings and to limit them.

The specification computed for our leading example is shown in figure 6.3. Note that, in contrast to the two axioms of Example 4, three axioms are computed. This is due to the unrolling of loops. Also note that the second and third computed axioms are instances of the second axiom of the intended specification.

Similarly to [1], due to bounded loop unrolling we cannot ensure completeness of the inferred specifications since we do not cover all possible execution paths. This is evident when comparing the automatically inferred axioms shown in the pattern above w.r.t. the expected specification given in Example 4. An effective generalization methodology is needed to properly cover all possible executions without incurring (hopefully) in significant loss of correctness.

7

Conclusions and Future Work

In this thesis we carried out the following contributions to the software engineering area: 1) we presented a technique for automated program specification inference adapting the fundamental concepts and the methodology of [1] to recent technology; 2) we implemented that technique in a prototype tool that automatically obtains axioms describing the input/output behavior of programs; and 3) we formulated a new specification for the semantics of a programming language, specifically KernelC, providing it with more complex capabilities such as lazy initialization and manipulation of structured data types. In the process, we also explored leading-edge techniques that show great potential for making software verification more simple and effective, and consequently improving the quality of software. Some of these techniques are: static analysis, symbolic execution, formal semantics and satisfiability problem solving, to name a few. We achieved to combine all of them to aim them towards the automated discovery of specifications through the analysis of real source code.

However, as part of a scientific research project, some improvements and extensions are being considered for future lines of investigation. We are currently working on defining a *generalization* algorithm that can distill more general axioms (such as the second axiom in Example 4) that we are not yet able to obtain. We follow the common synthesis approach that is based on using "skeletons" of generalizations, which are then refined to obtain a correct generalization of a set of axioms (w.r.t. the skeleton). The function that computes such skeletons basically induces them from iterations (loops and recursive calls) and is considered to be a parameter of the algorithm. Without entering into too much detail, candidate skeletons are given by a so-called "admissible template", that is, a non-ground $\mathbb{K}$ term that is used to guess the form that a given general axiom can have. A common drawback when resorting to skeletons is that the burden of defining/selecting the most suitable templates for a given problem usually rests with the user; hence usability is a key point that we cannot dismiss. Even if extensive research is still needed, our preliminary experiments reveal that axioms like the aforementioned more general one can be easily inferred automatically. Obviously, since we are using a threshold to stop loops, correctness cannot be ensured for all the general axioms that we compute, but they can still be useful for other verification processes or even be verified afterwards. A second, longer-term direction for research is to follow the abstraction-based, subsumption approach for symbolic execution of [2] to finitize symbolic execution while getting rid of any thresholds.

From the experimental point of view, there are certainly several ways that our prototype implementation can be improved. A refinement post-processing was defined in [1] that improves the quality of inferred specifications. Roughly speaking, when an observed pattern cannot be explained because its symbolic execution leads to final patterns that do not agree in the same result, the call pattern is (incrementally) split into multiple refined patterns until the considered observers eventually suffice to explain it. We plan to implement this refinement process in KindSpec 2.0 and measure the inference power gains. Actually, the main motivation of our work was not to improve efficiency w.r.t. [1], but rather to improve robustness, generality and mantainability.

Bibliography

[1] M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic Inference of Specifications using Matching Logic. In *Proc.e ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013*, pages 127–136. ACM, 2013.

[2] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):53–67, 2009.

[3] A. Arusoaie, D. Lucanu, V. Rusu, T.-F. Serbanuta, A. Stefanescu, and G. Roşu. Language Definitions as Rewrite Theories. In *10th International Workshop on Rewriting Logic and Its Applications (WRLA), Revised Selected Papers*, pages 97–112, 2014.

[4] G. Bacci, M. Comini, M. A. Feliú, and A. Villanueva. Automatic Synthesis of Specifications for First Order Curry Programs. In *Proc. of the 14th Intl. Symp. on ACM Principles and Practice of Declarative Programming (PPDP'12)*. ACM Press, 2012.

[5] M. Christodorescu, S. Jha, and C. Kruegel. Mining Specifications of Malicious Behavior. In *Proc. of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007)*, pages 5–14. ACM, 2007.

[6] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *Proc, 4th Int'l Conf. on Tests and Proofs (TAP 2010)*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.

[7] M. Clavel, F. Durán, S. Ejer, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[9] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proc. 30th International Conference on Software Engineering (ICSE 2008)*, pages 281–290. ACM, 2008.

[10] L. M. de Moura and B. Nikolaj. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[11] V. D'Silva, D. I. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[12] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

[13] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[14] S. Escobar, editor. *Rewriting Logic and Its Applications - 10th International Workshop, WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, volume 8663 of *Lecture Notes in Computer Science*. Springer, 2014.

[15] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing Intensional Behavior Models by Graph Transformation. In *Proc. 3st Int'l Conf. on Software Engineering (ICSE 2009)*, pages 430–440. IEEE, 2009.

[16] D. Giannakopoulou and C. S. Pasareanu. Interface Generation and Compositional Verification in JavaPathfinder. In *Proc. 12th In'l Conf. on Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2009.

[17] S. Hangal and M. S. Lam. Tracking down Software Bugs using Automatic Anomaly Detection. In *Proc. 22rd International Conference on Software Engineering (ICSE 2002)*, pages 291–301. ACM, 2002.

[18] J. Henkel and A. Diwan. Discovering Algebraic Specifications from Java Classes. In *Proc. ECOOP*, pages 431–456, 2003.

[19] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[20] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[21] Formal Systems Laboratory. Term rewriting and rewriting logic, November 2011. URL: `http://fsl.cs.illinois.edu/index.php/Term_Rewriting_and_Rewriting_Logic`.

[22] B. Liskov and J. Guttag. *Abstraction and specification in program development.* MIT Press, 1986.

[23] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. *Electr. Notes Theor. Comput. Sci.*, 4:190–225, 1996.

[24] C. S. Pasareanu and W. Visser. A Survey of new Trends in Symbolic Execution for Software Testing and Analysis. *STTT*, 11(4):339–353, 2009.

[25] G. Roşu. From Rewriting Logic, to Programming Language Semantics, to Program Verification. In *Logic, Rewriting, and Concurrency - Festschrift Symposium in Honor of José Meseguer*, Lecture Notes in Computer Science. Springer Verlag, 2015. To appear.

[26] G. Roşu, W. Schulte, and T.-F. Serbanuta. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009.

[27] G. Roşu and T.-F. Serbanuta. An Overview of the $\mathbb{K}$ Semantic Framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[28] G. Roşu and A. Stefanescu. Matching Logic: A New Program Verification Approach. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 868–871. ACM, 2011.

[29] M. Taghdiri and D.Jackson. Inferring Specifications to Detect Errors in Code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.

[30] N. Tillmann, F. Chen, and W. Schulte. Discovering Likely Method Specifications. In *Proc. 8th Int'l Conf. on Formal Engineering Methods (ICFEM 2006)*,

volume 4260 of *Lecture Notes in Computer Science*, pages 717–736. Springer, 2006.

[31] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. ISSTA 2002*, pages 218–228, 2002.

Appendix: Full KernelC Specification

KERNELC

```
MODULE KERNELC-SYNTAX

SYNTAX File ::= Globals

SYNTAX Globals ::= List{Global,""}

SYNTAX Global ::= StructDeclaration
                | FunctionDeclaration
                | FunctionDefinition
                | #include <stdio.h>
                | #include <stdlib.h>

SYNTAX FunctionDeclaration ::= Type Id(ParameterDeclarations) ;

SYNTAX FunctionDefinition ::= Type Id(ParameterDeclarations)StatementBlock

SYNTAX ParameterDeclarations ::= List{ParameterDeclaration,","}

SYNTAX ParameterDeclaration ::= Type Id

SYNTAX StructDeclaration ::= struct Id{VariableDeclarations} ;

SYNTAX VariableDeclarations ::= List{VariableDeclaration,""}

SYNTAX VariableDeclaration ::= Type Id ;

SYNTAX PrimitiveType ::= int
                       | void
                       | Type *

SYNTAX Type ::= PrimitiveType
              | struct Id

SYNTAX StatementBlock ::= {VariableDeclarations Statements}

SYNTAX Statements ::= List{Statement,""}

SYNTAX Statement ::= Expression = Expression ; [strict(2)]
                   | Expression += Expression ; [strict(2)]
```

```
     | Expression -= Expression ; [strict(2)]
     | Expression *= Expression ; [strict(2)]
     | Expression /= Expression ; [strict(2)]
     | Expression ++ ;
     | Expression - ;
     | Expression ; [strict]
     | if (Expression)Statement else Statement [avoid]
     | if (Expression)Statement
     | while (Expression)Statement
     | return Expression ; [strict(1)]
     | return ;
     | ;
     | StatementBlock
     | <stop>

SYNTAX Expression ::= Int
     | Id
     | NULL
     | (Expression) [bracket]
     | Expression . Id
     | Expression -> Id
     | Id(Arguments) [strict(2)]
     | sizeof (Type)
     | (Type)Expression [strict(2)]
     | - Expression [strict]
     | * Expression [strict]
     | & Expression
     | # Expression
     | Expression '
     | Expression * Expression [strict]
     | Expression / Expression [strict]
     | Expression + Expression [strict]
     | Expression - Expression [strict]
     | Expression < Expression [seqstrict]
     | Expression <= Expression [seqstrict]
     | Expression > Expression [seqstrict]
     | Expression >= Expression [seqstrict]
     | Expression == Expression [strict]
     | Expression != Expression [strict]
     | ! Expression
     | Expression && Expression [prefer, strict(1)]
     | Expression || Expression [strict(1)]

SYNTAX Arguments ::= List{Expression,“,”} [strict]

SYNTAX Id ::= main
     | malloc
     | free

END MODULE
```

MODULE KERNELC-SEMANTICS

RULE

$$\frac{G{:}Global\ Gs{:}Globals}{G \curvearrowright Gs}$$

[structural]

RULE

$$\frac{\bullet_{Globals}}{\bullet_{K}}$$

[structural]

RULE

$$\frac{\texttt{\#include <stdio.h>}}{\bullet_{K}}$$

[structural]

RULE

$$\frac{\texttt{\#include <stdlib.h>}}{\bullet_{K}}$$

[structural]

RULE

$$\frac{\text{—}{:}Type\ \ \text{—}{:}Id(\text{—}{:}ParameterDeclarations)\ \texttt{;}}{\bullet_{K}}$$

[structural]

RULE

k: $\frac{T{:}Type\ \ F{:}Id(PDs{:}ParameterDeclarations)SB{:}StatementBlock}{\bullet_{K}}$ ...

fun: ... $\frac{\bullet_{Map}}{F \mapsto \texttt{functionBody}\ (PDs, T, \{SB\ \texttt{return ;}\})}$

[structural]

RULE

k: $\frac{\texttt{struct}\ S{:}Id\{Members{:}VariableDeclarations\}\ \texttt{;}}{\texttt{makeMemberList}\ (Members, S)}$ ...

struct: ... $\frac{\bullet_{Map}}{S \mapsto \texttt{structFields}\ (\bullet_{List})}$

[structural]

RULE

$$\frac{S{:}Statement\ Ss{:}Statements}{S \curvearrowright Ss}$$

[structural]

RULE

$\frac{\bullet_{Statements}}{\bullet_{K}}$

[structural]

RULE

$\frac{VD{:}VariableDeclaration\ VDs{:}VariableDeclarations}{VD \curvearrowright VDs}$

[structural]

RULE

$\frac{\bullet_{VariableDeclarations}}{\bullet_{K}}$

[structural]

RULE

k: $\frac{T{:}PrimitiveType\ \ X{:}Id\ ;}{\bullet_{K}}$ ...

env: $\rho{:}Map\ \ \frac{\bullet_{Map}}{X \mapsto \texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(X))}$

heap: $Heap{:}Map\ \ \frac{\bullet_{Map}}{\texttt{pointer}\,(X) \mapsto \texttt{tv}\,(T,\ \texttt{undef})}$

locals: ... $\frac{\bullet_{List}}{\texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(X))}$

requires $\neg_{Bool} X\ \texttt{in keys}\,(\rho) \wedge_{Bool} \neg_{Bool}\ \texttt{pointer}\,(X)\ \texttt{in keys}\,(Heap)$

[structural]

RULE

k: $\frac{T{:}PrimitiveType\ \ X{:}Id\ ;}{\bullet_{K}}$ ...

env: ... $X \mapsto \frac{\texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(\xi{:}Expression))}{\texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(\#\,\xi))}$ ...

heap: ... $\frac{\bullet_{Map}}{\texttt{pointer}\,(\#\,\xi) \mapsto \texttt{tv}\,(T,\ \texttt{undef})}$

locals: ... $\frac{\bullet_{List}}{\texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(\#\,\xi))}$

[structural]

RULE

k: $\frac{T{:}PrimitiveType\ \ X{:}Id\ ;}{\bullet_{K}}$ ...

env: $\rho{:}Map\ \ \frac{\bullet_{Map}}{X \mapsto \texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(I))}$

heap: ... $\texttt{pointer}\,(X) \mapsto _\ \ _{:}Map\ \ \frac{\bullet_{Map}}{\texttt{pointer}\,(I{:}Id) \mapsto \texttt{tv}\,(T,\ \texttt{undef})}$

locals: ... $\frac{\bullet_{List}}{\texttt{tv}\,(T\,\texttt{*},\ \texttt{pointer}\,(I))}$

requires $\neg_{Bool} X\ \texttt{in keys}\,(\rho)$

[structural]

RULE

k: $\frac{\texttt{struct}\ S{:}Id\ \ X{:}Id\ ;}{\bullet_K}$

env: $\rho{:}Map\ \frac{\bullet_{Map}}{X \mapsto \texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(X))}$

heap: $Heap\ \frac{\bullet_{Map}}{\texttt{pointer}\,(X) \mapsto \texttt{tv}\,(\,\texttt{struct}\ S,\ \texttt{objectValues}\,(\,\texttt{initializeObject}\,(Fields)))}$

struct: $S \mapsto \texttt{structFields}\,(Fields{:}List)$

locals: $\frac{\bullet_{List}}{\texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(X))}$

requires $\neg_{Bool} X\ \texttt{in}\ \texttt{keys}\,(\rho) \wedge_{Bool} \neg_{Bool}\ \texttt{pointer}\,(X)\ \texttt{in}\ \texttt{keys}\,(Heap)$
[structural]

RULE

k: $\frac{\texttt{struct}\ S{:}Id\ \ X{:}Id\ ;}{\bullet_K}$

env: $X \mapsto \frac{\texttt{tv}\,(T\ \texttt{*},\ \texttt{pointer}\,(\xi{:}Expression))}{\texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(\,\texttt{\#}\ \xi))}$

heap: $\frac{\bullet_{Map}}{\texttt{pointer}\,(\,\texttt{\#}\ \xi) \mapsto \texttt{tv}\,(\,\texttt{struct}\ S,\ \texttt{objectValues}\,(\,\texttt{initializeObject}\,(Fields)))}$

struct: $S \mapsto \texttt{structFields}\,(Fields{:}List)$

locals: $\frac{\bullet_{List}}{\texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(\,\texttt{\#}\ \xi))}$

[structural]

RULE

k: $\frac{\texttt{struct}\ S{:}Id\ \ X{:}Id\ ;}{\bullet_K}$

env: $\rho{:}Map\ \frac{\bullet_{Map}}{X \mapsto \texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(I{:}Id))}$

heap: $\texttt{pointer}\,(X) \mapsto \text{—}\ \ \text{—}{:}Map\ \frac{\bullet_{Map}}{\texttt{pointer}\,(Id) \mapsto \texttt{tv}\,(\,\texttt{struct}\ S,\ \texttt{objectValues}\,(\,\texttt{initializeObject}\,(Fields)))}$

struct: $S \mapsto \texttt{structFields}\,(Fields{:}List)$

locals: $\frac{\bullet_{List}}{\texttt{tv}\,(\,\texttt{struct}\ S\ \texttt{*},\ \texttt{pointer}\,(I))}$

requires $\neg_{Bool} X\ \texttt{in}\ \texttt{keys}\,(\rho)$
[structural]

RULE

$\frac{\text{—}{:}TypedValue\ ;}{\bullet_K}$

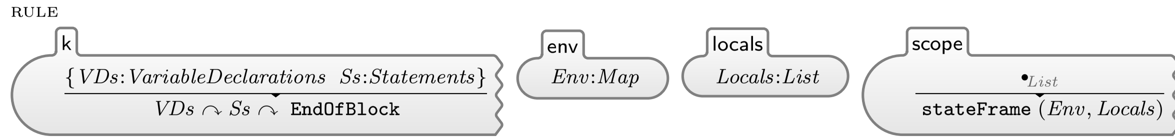
[structural]

RULE

$\frac{;}{\bullet_K}$

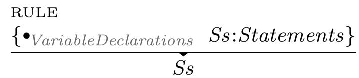
[structural]

RULE



requires $VDs \neq_K \bullet_{VariableDeclarations}$
[structural]

RULE

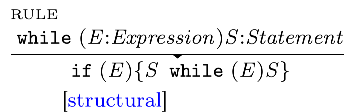


[structural]

RULE

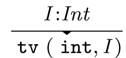
[structural]

RULE



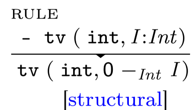
[structural]

RULE



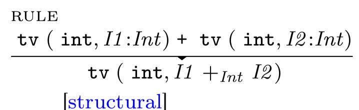
[structural]

RULE



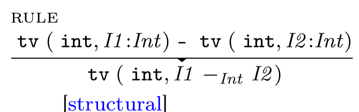
[structural]

RULE



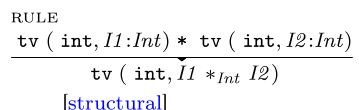
[structural]

RULE



[structural]

RULE



[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{/}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{tv}\ (\ \texttt{int}, \mathit{I1} \div_{\mathit{Int}} \mathit{I2})}$$

requires $\mathit{I2} =/=_{\mathit{Int}} \texttt{0}$
[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{<}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} <_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{<=}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} \leq_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{>}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} >_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{>=}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} \geq_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{==}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} ==_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (\ \texttt{int}, \mathit{I1}{:}\mathit{Int})\ \texttt{!=}\ \texttt{tv}\ (\ \texttt{int}, \mathit{I2}{:}\mathit{Int})}{\texttt{bool2int}\ (\mathit{I1} =/=_{\mathit{Int}} \mathit{I2})}$$

[structural]

RULE

$$\frac{\texttt{tv}\ (T{:}\mathit{Type}, \mathit{V1}{:}\mathit{Value})\ \texttt{==}\ \texttt{tv}\ (T, \mathit{V2}{:}\mathit{Value})}{\texttt{bool2int}\ (\mathit{V1} =_K \mathit{V2})}$$

requires $T \neq_K \texttt{int}$
[structural]

RULE

$$\frac{\texttt{tv}\ (T{:}\mathit{Type}, \mathit{V1}{:}\mathit{Value})\ \texttt{!=}\ \texttt{tv}\ (T, \mathit{V2}{:}\mathit{Value})}{\texttt{bool2int}\ (\mathit{V1} \neq_K \mathit{V2})}$$

requires $T \neq_K \texttt{int}$
[structural]

RULE

$\frac{\texttt{!}\ B{:}Bool}{\texttt{bool2int}\ (\neg_{Bool} B)}$

[structural]

RULE

$\frac{\texttt{sizeof (int)}}{\texttt{tv (int}, 1)}$

[structural]

RULE

$\frac{\texttt{sizeof}\ (T{:}Type\ \texttt{*})}{\texttt{tv (int}, 1)}$

[structural]

RULE

k: $\frac{\texttt{sizeof (struct}\ S{:}Id)}{\texttt{\#length}\ (Members)}$ struct: $S \mapsto \texttt{structFields}\ (Members{:}List)$

[structural]

RULE

$\frac{X{:}Id}{\texttt{* \&}\ X}$

[structural]

RULE

$\frac{\texttt{\& *}\ E{:}Expression}{E}$

[structural]

RULE

k: $\frac{\texttt{\&}\ X{:}Id}{TV}$ env: $X \mapsto TV{:}TypedValue$

[structural]

CONTEXT

$\texttt{\&}\ (\square\ \texttt{->}\ _)$

RULE

k: $\frac{\texttt{\& (tv (struct}\ S{:}Id\ \texttt{*},\ \texttt{pointer}\ (X{:}Expression))\ \texttt{->}\ F{:}Id)}{\texttt{tv}\ (T\ \texttt{*},\ \texttt{member}\ (X, F))}$ struct: $S \mapsto \texttt{structFields}\ (_{:}List\ \ \texttt{field}\ (F, T{:}Type)\ \ _{:}List)$

[structural]

RULE

malloc(tv (int, *Size*:*Int*))
⇒ CreateNewObject (*Size*)

[structural]

RULE

(*T*:*PrimitiveType* *) CreateNewObject (1)
⇒ NewSimplePointer

[structural]

RULE

k: tv (*T*:*Type* * *, pointer (*X*:*Expression*)) = NewSimplePointer ;
⇒ tv (*T* * *, pointer (*X*)) = tv (*T* *, pointer (* *X*)) ;

heap: $\bullet_{Map}$
⇒ pointer (* *X*) ↦ tv (*T*, undef)

locals: $\bullet_{List}$
⇒ tv (*T* *, pointer (* *X*))

[structural]

RULE

(struct *S*:*Id* *) CreateNewObject (*Size*:*Int*)
⇒ NewStructPointer (*Size*)

[structural]

RULE

k: tv (struct *S*:*Id* * *, pointer (*X*:*Expression*)) = NewStructPointer (*Size*:*Int*) ;
⇒ tv (struct *S* * *, pointer (*X*)) = tv (struct *S* *, pointer (* *X*)) ;

heap: $\bullet_{Map}$
⇒ pointer (* *X*) ↦ tv (struct *S*, objectValues (initializeObject (*Fields*)))

locals: $\bullet_{List}$
⇒ tv (struct *S* *, pointer (* *X*))

struct: *S* ↦ structFields (*Fields*:*List*)

[structural]

RULE

k: free(tv (*T*:*Type* *, pointer (*X*:*Expression*)))
⇒ cleanMemory (*H*, pointer (*X*))

heap: *H*:*Map*

[structural]

RULE

tv (*T1*:*Type* * *, —:*Pointer*) = tv (*T2*:*Type* * ⇒ *T1* *, null) ;

requires $T1 \neq_K T2$

[structural]

RULE

$\texttt{tv}\,(T1{:}Type\,\texttt{*}, \text{—}{:}Pointer)\ \texttt{==}\ \texttt{tv}\,(\underbrace{T2{:}Type\,\texttt{*}}_{T1\,\texttt{*}}, \texttt{null})$

requires $T1 \neq_K T2$

[structural]

RULE

$\texttt{tv}\,(\underbrace{T1{:}Type\,\texttt{*}}_{T2\,\texttt{*}}, \texttt{null})\ \texttt{==}\ \texttt{tv}\,(T2{:}Type\,\texttt{*}, \text{—}{:}Pointer)$

requires $T1 \neq_K T2$

[structural]

RULE

$\texttt{tv}\,(T1{:}Type\,\texttt{*}, \text{—}{:}Pointer)\ \texttt{!=}\ \texttt{tv}\,(\underbrace{T2{:}Type\,\texttt{*}}_{T1\,\texttt{*}}, \texttt{null})$

requires $T1 \neq_K T2$

[structural]

RULE

$\texttt{tv}\,(\underbrace{T1{:}Type\,\texttt{*}}_{T2\,\texttt{*}}, \texttt{null})\ \texttt{!=}\ \texttt{tv}\,(T2{:}Type\,\texttt{*}, \text{—}{:}Pointer)$

requires $T1 \neq_K T2$

[structural]

RULE

k: $\texttt{return tv}\,(\underbrace{T1{:}Type\,\texttt{*}}_{T2\,\texttt{*}}, \texttt{null})\ \texttt{;}$ ...

fun: ... $Called \mapsto \texttt{functionBody}\,(PDs, T2{:}Type\,\texttt{*}, \text{—}{:}StatementBlock)$ ...

current-function-call: $Called{:}Id(PDs{:}ParameterDeclarations)$

[structural]

RULE

$\dfrac{\texttt{tv}\,(\texttt{int}, 0)\ \texttt{\&\&}\ E{:}Expression}{\texttt{tv}\,(\texttt{int}, 0)}$

RULE

$\dfrac{\texttt{tv}\,(\texttt{int}, I{:}Int)\ \texttt{\&\&}\ E{:}Expression}{\texttt{bool2int}\,(\texttt{expression2Bool}\,(E))}$

requires $I =/=_{Int} 0$

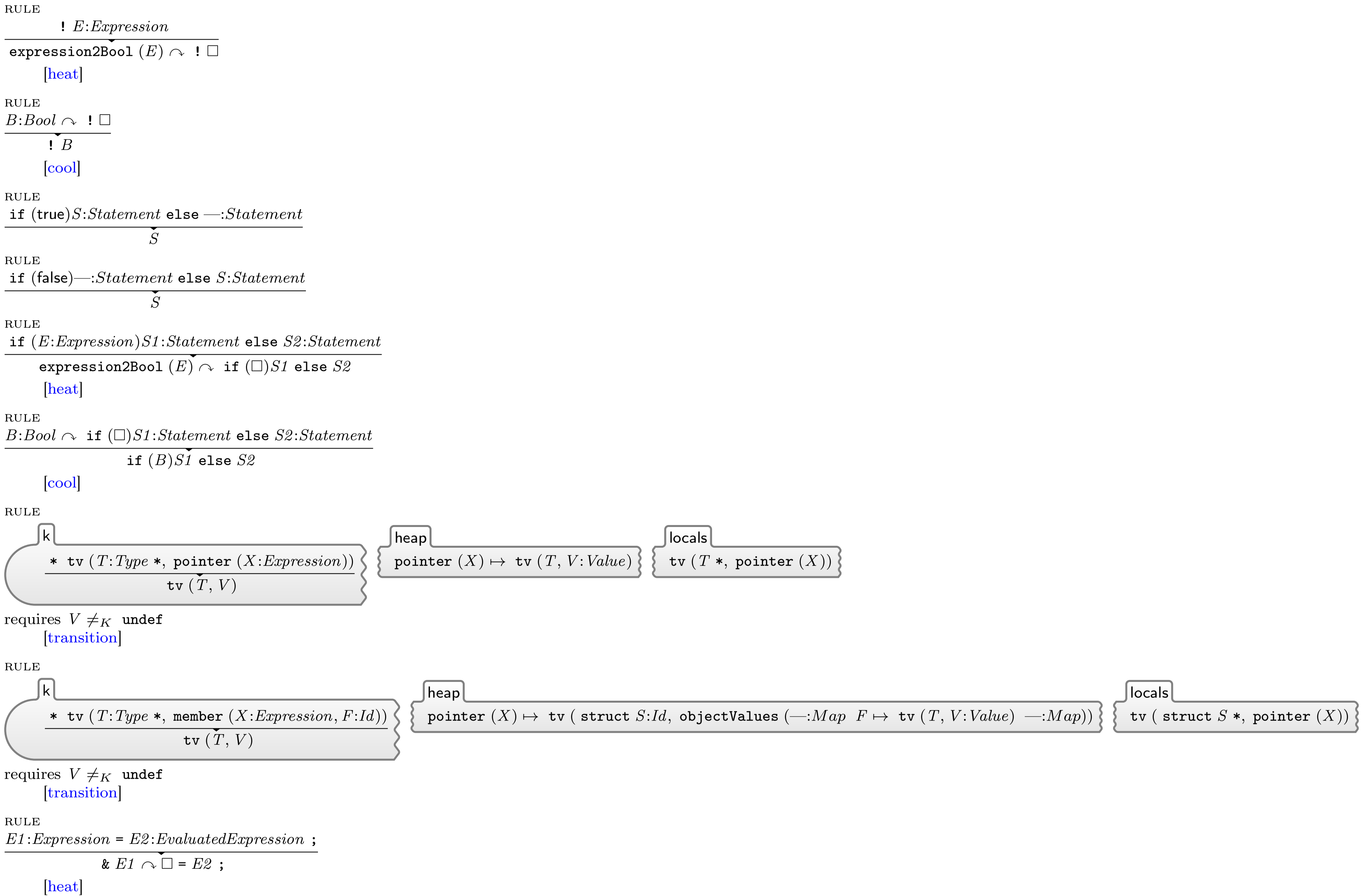
RULE

$\dfrac{\texttt{tv}\,(\texttt{int}, 0)\ \texttt{||}\ E{:}Expression}{\texttt{bool2int}\,(\texttt{expression2Bool}\,(E))}$

RULE

$\dfrac{\texttt{tv}\,(\texttt{int}, I{:}Int)\ \texttt{||}\ E{:}Expression}{\texttt{tv}\,(\texttt{int}, 1)}$

requires $I =/=_{Int} 0$


RULE
! E:Expression
expression2Bool (E) ↷ ! □
[heat]
RULE
B:Bool ↷ ! □
! B
[cool]
RULE
if (true)S:Statement else —:Statement
S
RULE
if (false)—:Statement else S:Statement
S
RULE
if (E:Expression)S1:Statement else S2:Statement
expression2Bool (E) ↷ if (□)S1 else S2
[heat]
RULE
B:Bool ↷ if (□)S1:Statement else S2:Statement
if (B)S1 else S2
[cool]
RULE
k
* tv (T:Type *, pointer (X:Expression))
tv (T, V)
heap
pointer (X) ↦ tv (T, V:Value)
locals
tv (T *, pointer (X))
requires V ≠K undef
[transition]
RULE
k
* tv (T:Type *, member (X:Expression, F:Id))
tv (T, V)
heap
pointer (X) ↦ tv (struct S:Id, objectValues (—:Map F ↦ tv (T, V:Value) —:Map))
locals
tv (struct S *, pointer (X))
requires V ≠K undef
[transition]
RULE
E1:Expression = E2:EvaluatedExpression ;
& E1 ↷ □ = E2 ;
[heat]

RULE

$TV{:}TypedValue \curvearrowright \square$ `=` $E2{:}EvaluatedExpression$ `;`

$\Rightarrow$ TV `=` $E2$ `;`

[cool]

RULE

k: `tv` $(T{:}Type$ `*`, `pointer` $(X{:}Expression))$ `=` `tv` $(T,\ V{:}Value)$ `;` $\Rightarrow$ $\bullet_K$

heap: `pointer` $(X) \mapsto$ `tv` $(T,\ \underline{\ \ }{:}Value)$ $\Rightarrow$ V

locals: `tv` $(T$ `*`, `pointer` $(X))$

[transition]

RULE

k: `tv` $(T{:}Type$ `*`, `member` $(X{:}Expression, F{:}Id))$ `=` `tv` $(T,\ V{:}Value)$ `;` $\Rightarrow$ $\bullet_K$

heap: `pointer` $(X) \mapsto$ `tv` (`struct` $S{:}Id$, `objectValues` $(\underline{\ \ }{:}Map\ \ F \mapsto$ `tv` $(T, \underline{\ \ }{:}Value)$ $\Rightarrow V$ $\ \underline{\ \ }{:}Map))$

locals: `tv` (`struct` S `*`, `pointer` $(X))$

requires $V \neq_K$ `undef`

[transition]

RULE

k: $Called{:}Id(EArgs{:}EvaluatedArguments) \curvearrowright K$ $\Rightarrow$ { `declareParameters` $(CalledPDs)$ `bindParameters` $(CalledPDs,\ EArgs,\ SB)$}

env: $Env{:}Map$ $\Rightarrow$ $\bullet_{Map}$

heap: $Heap{:}Map$

locals: $Locals{:}List$

fun: $Called \mapsto$ `functionBody` $(CalledPDs{:}ParameterDeclarations,\ \underline{\ \ }{:}Type,\ SB{:}StatementBlock)$

scope: $Scope{:}List$ $\Rightarrow$ $\bullet_{List}$

stack: $\Rightarrow$ `callStac`

requires $Called \neq_K$ malloc $\wedge_{Bool}$ $Called \neq_K$ free

RULE

k: $Called{:}Id(EArgs{:}EvaluatedArguments)$ $\Rightarrow$ { `declareParameters` (PDs) `bindParameters` $(PDs, EArgs, SB)$}

fun: $Called \mapsto$ `functionBody` $(PDs{:}ParameterDeclarations, \underline{\ \ }{:}Type, SB{:}StatementBlock)$

current-function-call: `no function` $\Rightarrow$ $Called(PDs)$

requires $Called \neq_K$ malloc $\wedge_{Bool}$ $Called \neq_K$ free

RULE

k: (`return` `tv` $(T{:}Type,\ V{:}Value)$ `;` $\curvearrowright$ `cleanAndRestoreMemory` $(H{:}Map, L{:}List) \curvearrowright K1)$ $\Rightarrow$ `cleanAndRestoreMemory` $(H, L) \curvearrowright$ `return` `tv` $(T,\ V)$ `;`

[structural]

RULE

k: (`return tv` (T:*Type*, V:*Value*) ;$\curvearrowright$ *K1*) ⇒ `return tv` (T, V) ;

requires $K1 \neq_K \bullet_K$
[structural]

RULE

k: `return tv` (T:*Type*, V:*Value*) ; ⇒ `tv` (T, V) $\curvearrowright$ `cleanAndRestoreMemory` (*CallerHeap*, *CallerLocals*) $\curvearrowright$ K

env: —:*Map* ⇒ *CallerEnv*

fun: *Called* $\mapsto$ `functionBody` (*CalledPDs*, T, —:*StatementBlock*)

scope: —:*List* ⇒ *CallerScope*

stack: `callStackFrame` (*Caller*:*FunctionProfile*, K, *CallerEnv*:*Map*, *CallerHeap*:*Map*, *CallerLocals*:*List*, *CallerScope*:*List*) ⇒ $\bullet_{List}$

current-function-call: *Called*:*Id*(*CalledPDs*:*ParameterDeclarations*) ⇒ *Caller*

RULE

k: `return tv` (T:*Type*, V:*Value*) ; ⇒ `tv` (T, V)

fun: *Called* $\mapsto$ `functionBody` (*PDs*:*ParameterDeclarations*, T, —:*StatementBlock*)

stack: $\bullet_{List}$

scope: —:*List* ⇒ $\bullet_{List}$

current-function-call: *Called*:*Id*(*PDs*:*ParameterDeclarations*)

requires $PDs \neq_K \bullet_{ParameterDeclarations}$

RULE

k: `return tv` (T:*Type*, V:*Value*) ; ⇒ `tv` (T, V)

fun: *Called* $\mapsto$ `functionBody` (—:*ParameterDeclarations*, T, —:*StatementBlock*)

stack: $\bullet_{List}$

scope: —:*List* ⇒ $\bullet_{List}$

current-function-call: *Called*:*Id*($\bullet_{ParameterDeclarations}$) ⇒ *Called*()

END MODULE

MODULE KERNELC-AUXILIARY

SYNTAX *StatementBlock* ::= {*Statements*}

RULE
{*Ss*:*Statements*}
{•*VariableDeclarations* *Ss*}
[structural]

RULE
`if` (*E*:*Expression*)*S*:*Statement*
`if` (*E*)*S* `else` ;
[structural]

RULE
E1:*Expression* `+=` *E2*:*Expression* ;
E1 `=` *E1* `+` *E2* ;
[structural]

RULE
E1:*Expression* `-=` *E2*:*Expression* ;
E1 `=` *E1* `-` *E2* ;
[structural]

RULE
E1:*Expression* `*=` *E2*:*Expression* ;
E1 `=` *E1* `*` *E2* ;
[structural]

RULE
E1:*Expression* `/=` *E2*:*Expression* ;
E1 `=` *E1* `/` *E2* ;
[structural]

RULE
E:*Expression* `++` ;
E `=` *E* `+` `1` ;
[structural]

RULE
E:*Expression* `-` ;
E `=` *E* `-` `1` ;
[structural]

RULE
`return` ;
`return` `tv` (`void`, `undef`) ;
[structural]

RULE
`NULL`
`tv` (`void` `*`, `null`)
[structural]

RULE

$$\frac{E{:}Expression\ \texttt{->}\ F{:}Id}{\texttt{* \&}\ (E\ \texttt{->}\ F)}$$

[structural]

SYNTAX *Expression* ::= *EvaluatedExpression*

SYNTAX *EvaluatedExpression* ::= *TypedValue*
| *Bool*
| *String*

SYNTAX *Value* ::= *Int*
| *Pointer*
| `undef`
| `objectValues` (*Map*)

SYNTAX *Pointer* ::= `pointer` (*Expression*)
| `member` (*Expression*, *Id*)
| `null`

SYNTAX *EvaluatedArguments* ::= *List*{*EvaluatedExpression*,“,”}

SYNTAX *FunctionProfile* ::= `no function`
| *Id*(*ParameterDeclarations*)
| *Id*()

SYNTAX *TypedValue* ::= `tv` (*Type*, *Value*)

SYNTAX *KItem* ::= `functionBody` (*ParameterDeclarations*, *Type*, *StatementBlock*)
| `callStackFrame` (*FunctionProfile*, *K*, *Map*, *Map*, *List*, *List*)
| `structFields` (*List*)
| `field` (*Id*, *Type*)
| `stateFrame` (*Map*, *List*)

SYNTAX *Statement* ::= `EndOfBlock`
| `cleanMemory` (*Map*, *Pointer*)
| `cleanAndRestoreMemory` (*Map*, *List*)
| `garbageCollect` (*Map*)
| `restoreMemory` (*Map*, *List*)

SYNTAX *Expression* ::= `eliminatePointer` (*Pointer*)

RULE

k:
$$\frac{\texttt{cleanMemory}\ (\ \texttt{pointer}\ (Y{:}Expression) \mapsto\ \texttt{tv}\ (T{:}Type,\ \texttt{pointer}\ (X{:}Expression))\ \ H{:}Map,\ \texttt{pointer}\ (X))}{\texttt{cleanMemory}\ (H,\ \texttt{pointer}\ (X))}$$

heap:
$$\texttt{pointer}\ (Y) \mapsto \frac{__}{\texttt{tv}\ (T,\ \texttt{undef})}$$

[structural]

RULE

$$\frac{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(\text{—}{:}\mathit{Expression}) \mapsto \texttt{tv}\,(\text{—}{:}\mathit{Type},\, V{:}\mathit{Value})\;\; H{:}\mathit{Map},\, \texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{cleanMemory}\,(H,\, \texttt{pointer}\,(X))}$$

requires $V \neq_K$ pointer (X)
[structural]

RULE

k:
$$\frac{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(Y{:}\mathit{Expression}) \mapsto \texttt{tv}\,(\,\texttt{struct}\; S{:}\mathit{Id},\, \texttt{objectValues}\,(F{:}\mathit{Id} \mapsto \texttt{tv}\,(T{:}\mathit{Type},\, \texttt{pointer}\,(X{:}\mathit{Expression}))\;\; OV{:}\mathit{Map}))\;\; H{:}\mathit{Map},\, \texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(Y) \mapsto \texttt{tv}\,(\,\texttt{struct}\; S,\, \texttt{objectValues}\,(OV))\;\; H,\, \texttt{pointer}\,(X))}$$

heap:
$$\texttt{pointer}\,(Y) \mapsto \texttt{tv}\,(\,\texttt{struct}\; S,\, \texttt{objectValues}\,(\text{—}{:}\mathit{Map}\;\; F \mapsto \frac{\text{—}}{\texttt{tv}\,(T,\, \texttt{undef})}\;\; \text{—}{:}\mathit{Map}))$$

[structural]

RULE

$$\frac{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(Y{:}\mathit{Expression}) \mapsto \texttt{tv}\,(\,\texttt{struct}\; S{:}\mathit{Id},\, \texttt{objectValues}\,(\text{—}{:}\mathit{Id} \mapsto \texttt{tv}\,(\text{—}{:}\mathit{Type},\, V{:}\mathit{Value})\;\; OV{:}\mathit{Map}))\;\; H{:}\mathit{Map},\, \texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(Y) \mapsto \texttt{tv}\,(\,\texttt{struct}\; S,\, \texttt{objectValues}\,(OV))\;\; H,\, \texttt{pointer}\,(X))}$$

requires $V \neq_K$ pointer (X)
[structural]

RULE

$$\frac{\texttt{cleanMemory}\,(\,\texttt{pointer}\,(\text{—}{:}\mathit{Expression}) \mapsto \texttt{tv}\,(\,\texttt{struct}\; \text{—}{:}\mathit{Id},\, \texttt{objectValues}\,(\bullet_{\mathit{Map}}))\;\; H{:}\mathit{Map},\, \texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{cleanMemory}\,(H,\, \texttt{pointer}\,(X))}$$

[structural]

RULE

$$\frac{\texttt{cleanMemory}\,(\bullet_{\mathit{Map}},\, \texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{eliminatePointer}\,(\,\texttt{pointer}\,(X))}$$

[structural]

RULE

k:
$$\frac{\texttt{eliminatePointer}\,(\,\texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{tv}\,(\,\texttt{void},\, \texttt{undef})}$$

heap:
$$\frac{(\,\texttt{pointer}\,(X) \mapsto \text{—})}{\bullet_{\mathit{Map}}}$$

locals:
$$\frac{\texttt{tv}\,(\text{—}{:}\mathit{Type},\, \texttt{pointer}\,(X))}{\bullet_{\mathit{List}}}$$

[structural]

RULE

k:
$$\frac{\texttt{eliminatePointer}\,(\,\texttt{pointer}\,(X{:}\mathit{Expression}))}{\texttt{tv}\,(\,\texttt{void},\, \texttt{undef})}$$

heap:
$$\frac{(\,\texttt{pointer}\,(X) \mapsto \text{—})}{\bullet_{\mathit{Map}}}$$

locals: $\mathit{Locals}{:}\mathit{List}$

requires $\neg_{Bool}$ pointer (X) in listPointers (Locals)
[structural]

RULE

k: $\dfrac{(TV{:}TypedValue \curvearrowright \texttt{cleanAndRestoreMemory}\ (H{:}Map, L{:}List) \curvearrowright S \curvearrowright Ks)}{(TV \curvearrowright S \curvearrowright \texttt{cleanAndRestoreMemory}\ (H, L) \curvearrowright Ks)}$

[structural]

RULE

k: $\dfrac{\texttt{cleanAndRestoreMemory}\ (PreviousHeap{:}Map, PreviousLocals{:}List)}{(\texttt{garbageCollect}\ (CurrentHeap) \curvearrowright \texttt{restoreMemory}\ (PreviousHeap, PreviousLocals))}$ … heap: $CurrentHeap{:}Map$

[structural]

RULE

k: $\dfrac{\texttt{garbageCollect}\ (\texttt{pointer}\ (X{:}Expression) \mapsto \text{—}\ \ HeapCopy{:}Map)}{\texttt{garbageCollect}\ (HeapCopy)}$ … env: $E1{:}Map\ \ \text{—}{:}Id \mapsto \texttt{tv}\ (\text{—}, \texttt{pointer}\ (X))\ \ E2{:}Map$

[structural]

RULE

k: $\dfrac{\texttt{garbageCollect}\ (\texttt{pointer}\ (X{:}Expression) \mapsto \text{—}\ \ HeapCopy{:}Map)}{\texttt{garbageCollect}\ (HeapCopy)}$ … heap: $H1{:}Map\ \ \texttt{pointer}\ (\text{—}{:}Expression) \mapsto \texttt{tv}\ (\text{—}, \texttt{pointer}\ (X))\ \ H2{:}Map$

[structural]

RULE

k: $\dfrac{\texttt{garbageCollect}\ (\texttt{pointer}\ (X{:}Expression) \mapsto \text{—}\ \ HeapCopy{:}Map)}{\texttt{garbageCollect}\ (HeapCopy)}$ … heap: $H1{:}Map\ \ \texttt{pointer}\ (\text{—}{:}Expression) \mapsto \texttt{tv}\ (\texttt{struct}\ \text{—}{:}Id,\ \texttt{objectValues}\ (O1{:}Map\ \ \text{—}{:}Id \mapsto \texttt{tv}\ (\text{—}, \texttt{pointer}\ (X))\ \ O2{:}Map))\ \ H2{:}Map$

[structural]

RULE

k: $\dfrac{\texttt{garbageCollect}\ (\texttt{pointer}\ (X{:}Expression) \mapsto \texttt{tv}\ (T{:}Type, \text{—})\ \ HeapCopy{:}Map)}{(\texttt{eliminatePointer}\ (\texttt{pointer}\ (X))\ ; \curvearrowright \texttt{garbageCollect}\ (HeapCopy))}$ … env: $Env{:}Map$ heap: $Heap{:}Map$

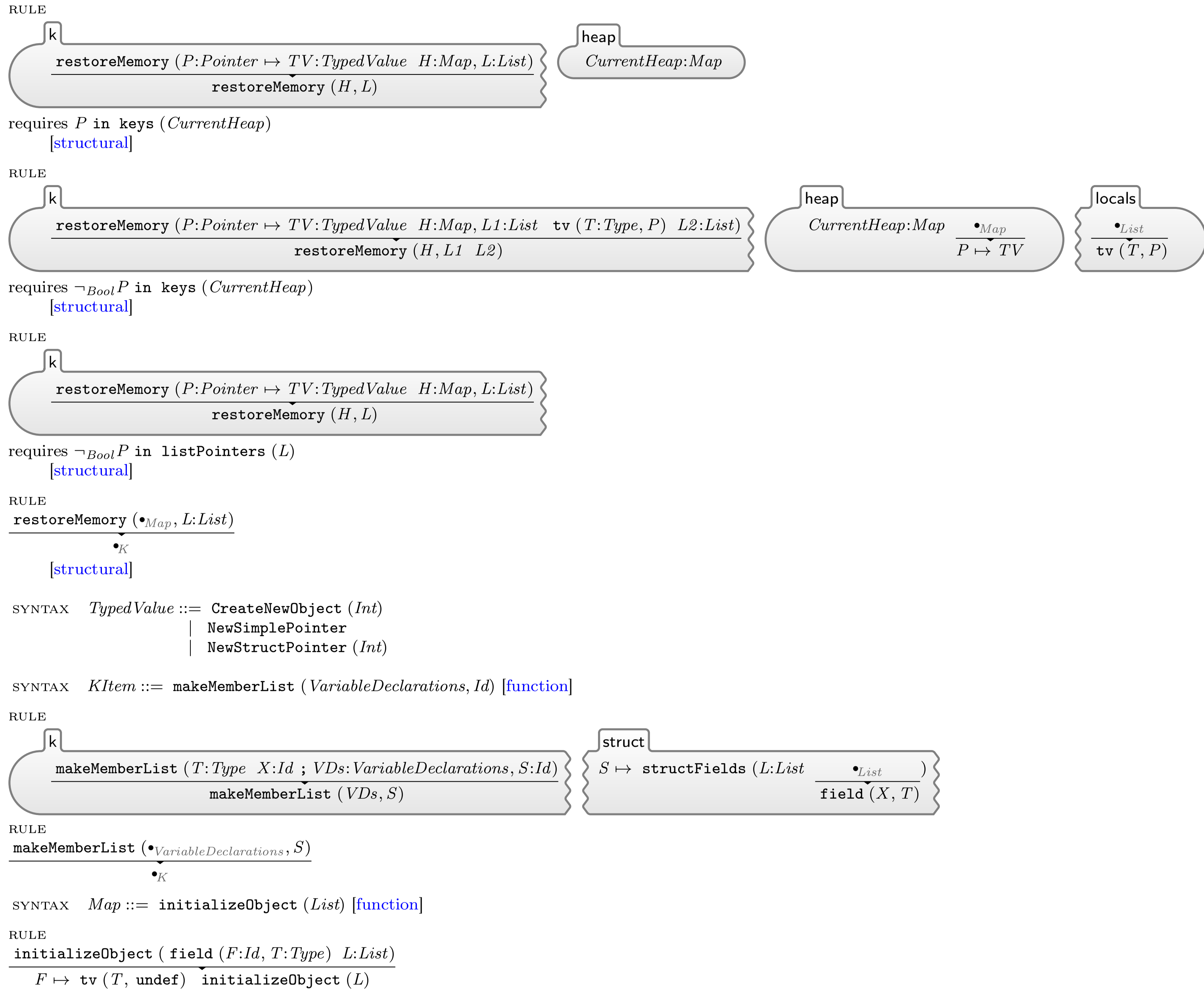
requires $\neg_{Bool}\ \texttt{tv}\ (T\ \texttt{*}, \texttt{pointer}\ (X))\ \texttt{in}\ \texttt{values}\ (Env) \wedge_{Bool} \neg_{Bool}\ \texttt{tv}\ (T\ \texttt{*}, \texttt{pointer}\ (X))\ \texttt{in}\ \texttt{values}\ (Heap) \wedge_{Bool} \neg_{Bool}\ \texttt{tv}\ (T\ \texttt{*}, \texttt{pointer}\ (X))\ \texttt{in}\ \texttt{listObjectValues}\ (Heap)$

[structural]

RULE

$\dfrac{\texttt{garbageCollect}\ (\bullet_{Map})}{\bullet_{K}}$

[structural]



RULE
k
restoreMemory (P:Pointer ↦ TV:TypedValue H:Map, L:List)
restoreMemory (H, L)
heap
CurrentHeap:Map
requires P in keys (CurrentHeap)
[structural]
RULE
k
restoreMemory (P:Pointer ↦ TV:TypedValue H:Map, L1:List tv (T:Type, P) L2:List)
restoreMemory (H, L1 L2)
heap
CurrentHeap:Map •Map
P ↦ TV
locals
•List
tv (T, P)
requires ¬Bool P in keys (CurrentHeap)
[structural]
RULE
k
restoreMemory (P:Pointer ↦ TV:TypedValue H:Map, L:List)
restoreMemory (H, L)
requires ¬Bool P in listPointers (L)
[structural]
RULE
restoreMemory (•Map, L:List)
•K
[structural]
SYNTAX TypedValue ::= CreateNewObject (Int)
| NewSimplePointer
| NewStructPointer (Int)
SYNTAX KItem ::= makeMemberList (VariableDeclarations, Id) [function]
RULE
k
makeMemberList (T:Type X:Id ; VDs:VariableDeclarations, S:Id)
makeMemberList (VDs, S)
struct
S ↦ structFields (L:List •List)
field (X, T)
RULE
makeMemberList (•VariableDeclarations, S)
•K
SYNTAX Map ::= initializeObject (List) [function]
RULE
initializeObject (field (F:Id, T:Type) L:List)
F ↦ tv (T, undef) initializeObject (L)

RULE

`initializeObject` ($\bullet_{List}$)
⇒ $\bullet_{Map}$

SYNTAX *VariableDeclarations* ::= `declareParameters` (*ParameterDeclarations*) [function]

RULE

`declareParameters` (T:*Type* X:*Id* , PDs:*ParameterDeclarations*)
⇒ T X ; `declareParameters` (PDs)

RULE

`declareParameters` ($\bullet_{ParameterDeclarations}$)
⇒ $\bullet_{VariableDeclarations}$

SYNTAX *Statements* ::= `bindParameters` (*ParameterDeclarations*, *EvaluatedArguments*, *StatementBlock*) [function]

RULE

`bindParameters` ((—:*Type* X:*Id* , PDs:*ParameterDeclarations*), (TV:*TypedValue* , $EArgs$:*EvaluatedArguments*), SB:*StatementBlock*)
⇒ X = TV ; `bindParameters` (PDs, $EArgs$, SB)

RULE

`bindParameters` ($\bullet_{ParameterDeclarations}$, $\bullet_{EvaluatedArguments}$, SB:*StatementBlock*)
⇒ SB $\bullet_{Statements}$

SYNTAX *KItem* ::= `bool2int` (*Bool*) [strict]

RULE

`bool2int` (true)
⇒ `tv` (`int`, 1)

RULE

`bool2int` (false)
⇒ `tv` (`int`, 0)

SYNTAX *KItem* ::= `expression2Bool` (*Expression*) [strict]

RULE

`expression2Bool` (`tv` (`int`, I:*Int*))
⇒ $I =/=_{Int} 0$

[structural]

RULE

`expression2Bool` (`tv` (T:*Type* `*`, P:*Pointer*))
⇒ $P \neq_K$ `null`

[structural]

SYNTAX *Id* ::= `Expression2Id` (*Expression*) [function]

RULE

`Expression2Id` (I:*Int*)
⇒ `String2Id` (`Int2String` (I))

[structural]

RULE

$$\frac{\texttt{Expression2Id}\,(I{:}Id)}{I}$$

[structural]

RULE

$$\frac{\texttt{Expression2Id}\,(\,\texttt{*}\ E{:}Expression)}{\texttt{Expression2Id}\,(E)}$$

[structural]

RULE

$$\frac{\texttt{Expression2Id}\,(E{:}Expression\ \texttt{.}\ I{:}Id)}{\texttt{String2Id}\,(\,\texttt{Id2String}\,(\,\texttt{Expression2Id}\,(E)) +_{String}\ \texttt{Id2String}\,(I))}$$

[structural]

SYNTAX *TypedValue* ::= `#length` (*Map*) [function]
| `#length` (*List*) [function]
| `listLength` (*List*, *TypedValue*) [function]
| `mapLength` (*Map*, *TypedValue*) [function]

RULE

$$\frac{\texttt{\#length}\,(M{:}Map)}{\texttt{mapLength}\,(M,\ \texttt{tv}\,(\,\texttt{int}, 0))}$$

[structural]

RULE

$$\frac{\texttt{mapLength}\,(\text{—} \mapsto \text{—}\ \ M{:}Map,\ \texttt{tv}\,(\,\texttt{int}, Len))}{\texttt{mapLength}\,(M,\ \texttt{tv}\,(\,\texttt{int}, Len +_{Int} 1))}$$

[structural]

RULE

$$\frac{\texttt{mapLength}\,(\bullet_{Map},\ \texttt{tv}\,(\,\texttt{int}, Len))}{\texttt{tv}\,(\,\texttt{int}, Len)}$$

[structural]

RULE

$$\frac{\texttt{\#length}\,(L{:}List)}{\texttt{listLength}\,(L,\ \texttt{tv}\,(\,\texttt{int}, 0))}$$

[structural]

RULE

$$\frac{\texttt{listLength}\,(\text{—}\ \ L{:}List,\ \texttt{tv}\,(\,\texttt{int}, Len))}{\texttt{listLength}\,(L,\ \texttt{tv}\,(\,\texttt{int}, Len +_{Int} 1))}$$

[structural]

RULE

$$\frac{\texttt{listLength}\,(\bullet_{List},\ \texttt{tv}\,(\,\texttt{int}, Len))}{\texttt{tv}\,(\,\texttt{int}, Len)}$$

[structural]

SYNTAX $Bool ::= Pointer$ `==` $Pointer$
$\mid$ $Pointer$ `=/=` $Pointer$

SYNTAX $Set ::=$ `listObjectValues` (Map) [function]
$\mid$ `listObjectValuesFun` (Map, Set) [function]
$\mid$ `listPointers` $(List)$ [function]
$\mid$ `listPointersFun` $(List, Set)$ [function]

RULE
`listObjectValues` $(H{:}Map)$
―――
`listObjectValuesFun` $(H, \bullet_{Set})$
[structural]

RULE
`listObjectValuesFun` $(\text{—} \mapsto$ `tv` $(\text{—}, I{:}Int)\ \ H{:}Map, S{:}Set)$
―――
`listObjectValuesFun` (H, S)
[structural]

RULE
`listObjectValuesFun` $(\text{—} \mapsto$ `tv` $(\text{—}, P{:}Pointer)\ \ H{:}Map, S{:}Set)$
―――
`listObjectValuesFun` (H, S)
[structural]

RULE
`listObjectValuesFun` $(\text{—} \mapsto$ `tv` $(\text{—},$ `undef`$)\ \ H{:}Map, S{:}Set)$
―――
`listObjectValuesFun` (H, S)
[structural]

RULE
`listObjectValuesFun` $(P{:}Pointer \mapsto$ `tv` (`struct` $TS{:}Id$, `objectValues` $(\text{—} \mapsto TV{:}TypedValue\ \ OV{:}Map))\ \ H{:}Map, S{:}Set)$
―――
`listObjectValuesFun` $(P \mapsto$ `tv` (`struct` TS, `objectValues` $(OV))\ \ H, (TV\ \ S))$
[structural]

RULE
`listObjectValuesFun` $(\text{—} \mapsto$ `tv` (`struct` $\text{—}{:}Id$, `objectValues` $(\bullet_{Map}))\ \ H{:}Map, S{:}Set)$
―――
`listObjectValuesFun` (H, S)
[structural]

RULE
`listObjectValuesFun` $(\bullet_{Map}, S{:}Set)$
―――
S
[structural]

RULE
`listPointers` $(L{:}List)$
―――
`listPointersFun` $(L, \bullet_{Set})$
[structural]

RULE

$$\frac{\texttt{listPointersFun}\,(\,\texttt{tv}\,(T{:}Type, P{:}Pointer)\ \ L{:}List, S{:}Set)}{\texttt{listPointersFun}\,(L, (P\ \ S))}$$

[structural]

RULE

$$\frac{\texttt{listPointersFun}\,(\bullet_{List}, S{:}Set)}{S}$$

[structural]

END MODULE

MODULE KERNELC-EXCEPTION

SYNTAX *Exception* ::= EXCEPTION: DIVISION BY ZERO
| EXCEPTION: NULL POINTER ACCESS
| SYNTAX ERROR: INCORRECT RETURN TYPE
| SYNTAX ERROR: THE FUNCTION CALLED DOES NOT EXIST

RULE

k: $$\frac{(T1{:}TypedValue\ \texttt{/}\ \texttt{tv}\,(\,\texttt{int}, 0) \curvearrowright K)}{\texttt{EXCEPTION: DIVISION BY ZERO}}$$

[structural]

RULE

k: $$\frac{(\,\texttt{*}\ \texttt{tv}\,(T{:}Type\ \texttt{*},\ \texttt{null}) \curvearrowright K)}{\texttt{EXCEPTION: NULL POINTER ACCESS}}$$

[structural]

RULE

k: $$\frac{(\,\texttt{tv}\,(T{:}Type\ \texttt{*},\ \texttt{null}) = \text{—}\ ; \curvearrowright K)}{\texttt{EXCEPTION: NULL POINTER ACCESS}}$$

[structural]

RULE

k: $$\frac{(\,\texttt{\&}\,(\,\texttt{tv}\,(\,\texttt{struct}\ S{:}Id\ \texttt{*},\ \texttt{null})\ \texttt{->}\ F{:}Id) \curvearrowright K)}{\texttt{EXCEPTION: NULL POINTER ACCESS}}$$

[structural]

RULE

k: $\dfrac{(\texttt{return tv}\ (T1{:}Type, —)\ ; \curvearrowright K)}{\texttt{SYNTAX ERROR: INCORRECT RETURN TYPE}}$

current-function-call: $F{:}Id(PDs{:}ParameterDeclarations)$

fun: $F \mapsto \texttt{functionBody}\ (PDs, T2{:}Type, —{:}StatementBlock)$

requires $T1 \neq_K T2$
[structural]

END MODULE

MODULE KERNELC-SYMBOLIC

SYNTAX $Arguments ::=$ `Symbolic`

SYNTAX $KItem ::=$ `makeSymbolic` $(ParameterDeclarations)$ [function]

RULE

k: $\dfrac{F{:}Id(\texttt{Symbolic})}{\texttt{makeSymbolic}\ (PDs) \curvearrowright \bullet_{EvaluatedArguments} \curvearrowright F(\bullet_{EvaluatedArguments})}$

fun: $F \mapsto \texttt{functionBody}\ (PDs{:}ParameterDeclarations, —{:}Type, —{:}StatementBlock)$

[structural]

RULE
$$\frac{\texttt{makeSymbolic}\ (T{:}Type\ \ X{:}Id\,,\ PDs{:}ParameterDeclarations) \curvearrowright SymArgs{:}EvaluatedArguments}{\texttt{makeSymbolic}\ (PDs) \curvearrowright \texttt{tv}\ (\texttt{int}, \#symInt(X)),\ SymArgs}$$
requires $T =_K$ `int`

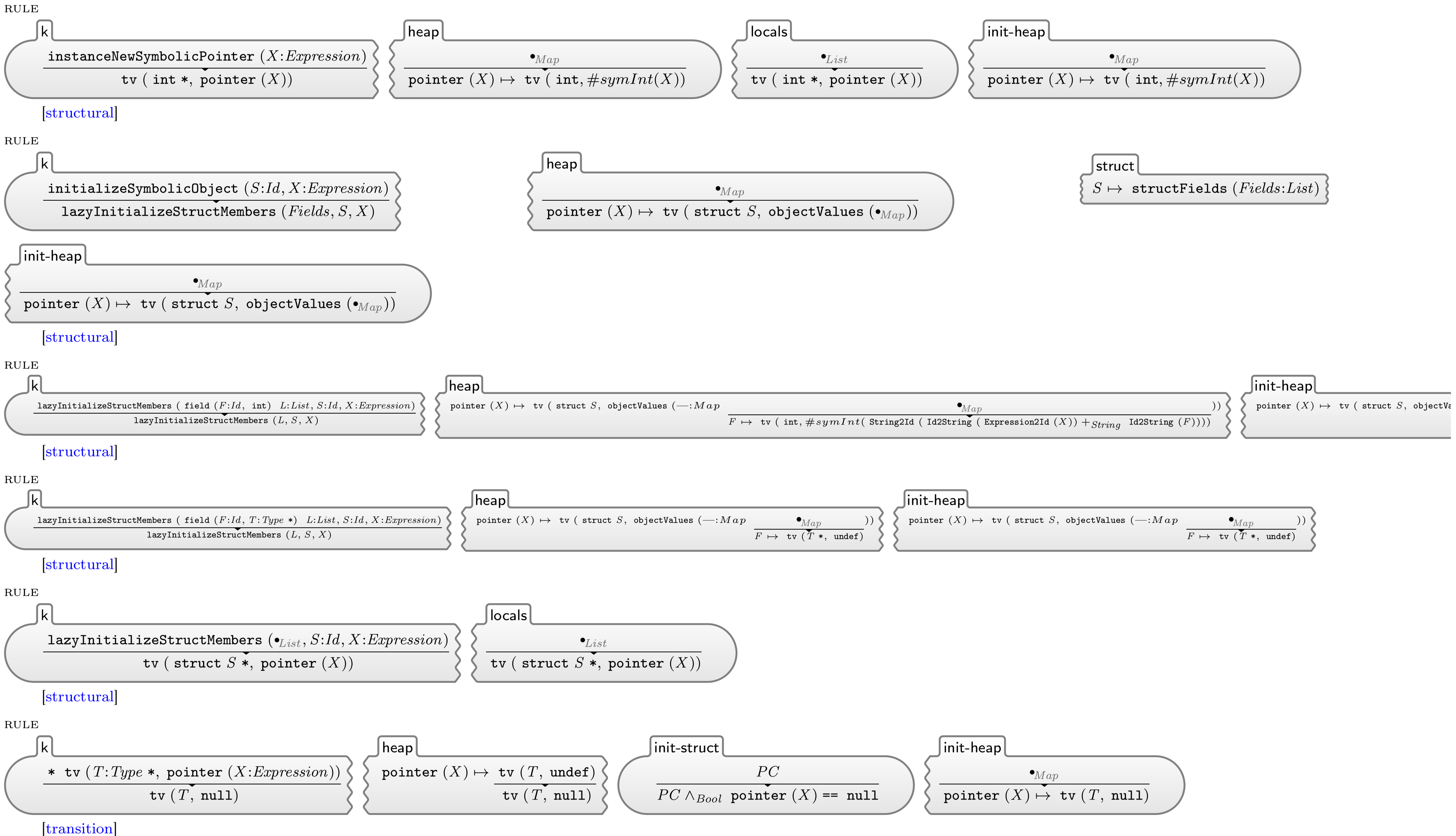
RULE
$$\frac{\texttt{makeSymbolic}\ (T{:}Type\ \ X{:}Id\,,\ PDs{:}ParameterDeclarations) \curvearrowright SymArgs{:}EvaluatedArguments}{\texttt{makeSymbolic}\ (PDs) \curvearrowright \texttt{tv}\ (T,\ \texttt{undef}),\ SymArgs}$$
requires $T \neq_K$ `int`

RULE
$$\frac{\texttt{makeSymbolic}\ (\bullet_{ParameterDeclarations})}{\bullet_K}$$

RULE
$$\frac{S{:}EvaluatedExpression\,,\ SymArgs{:}EvaluatedArguments \curvearrowright F{:}Id(Args{:}EvaluatedArguments)}{SymArgs \curvearrowright F(S\,,\ Args)}$$

RULE
$$\frac{\bullet_{EvaluatedArguments} \curvearrowright F{:}Id(Args{:}EvaluatedArguments)}{F(Args)}$$

SYNTAX $KItem ::=$ `instanceNewSymbolicPointer` $(Expression)$ [function]
| `initializeSymbolicObject` $(Id, Expression)$ [function]
| `lazyInitializeStructMembers` $(List, Id, Expression)$ [function]


RULE
k
instanceNewSymbolicPointer (X:Expression)
tv (int *, pointer (X))
heap
•Map
pointer (X) ↦ tv (int, #symInt(X))
locals
•List
tv (int *, pointer (X))
init-heap
•Map
pointer (X) ↦ tv (int, #symInt(X))
[structural]
RULE
k
initializeSymbolicObject (S:Id, X:Expression)
lazyInitializeStructMembers (Fields, S, X)
heap
•Map
pointer (X) ↦ tv (struct S, objectValues (•Map))
struct
S ↦ structFields (Fields:List)
init-heap
•Map
pointer (X) ↦ tv (struct S, objectValues (•Map))
[structural]
RULE
k
lazyInitializeStructMembers (field (F:Id, int) L:List, S:Id, X:Expression)
lazyInitializeStructMembers (L, S, X)
heap
pointer (X) ↦ tv (struct S, objectValues (—:Map
•Map
F ↦ tv (int, #symInt(String2Id (Id2String (Expression2Id (X)) +String Id2String (F))))
init-heap
pointer (X) ↦ tv (struct S, objectVa
[structural]
RULE
k
lazyInitializeStructMembers (field (F:Id, T:Type *) L:List, S:Id, X:Expression)
lazyInitializeStructMembers (L, S, X)
heap
pointer (X) ↦ tv (struct S, objectValues (—:Map
•Map
F ↦ tv (T *, undef)
init-heap
pointer (X) ↦ tv (struct S, objectValues (—:Map
•Map
F ↦ tv (T *, undef)
[structural]
RULE
k
lazyInitializeStructMembers (•List, S:Id, X:Expression)
tv (struct S *, pointer (X))
locals
•List
tv (struct S *, pointer (X))
[structural]
RULE
k
* tv (T:Type *, pointer (X:Expression))
tv (T, null)
heap
pointer (X) ↦ tv (T, undef)
tv (T, null)
init-struct
PC
PC ∧Bool pointer (X) == null
init-heap
•Map
pointer (X) ↦ tv (T, null)
[transition]

RULE

k: * tv (T:*Type* *, member (S:*Expression*, F:*Id*)) ⇒ tv (T, null)

heap: pointer (S) ↦ tv (struct ST:*Id*, objectValues (*FM1*:*Map* F ↦ tv (T, undef) ⇒ tv (T, null) *FM2*:*Map*))

init-struct: PC ⇒ $PC \wedge_{Bool}$ pointer (S . F) == null

init-heap: ρ:*Map* ⇒ pointer (S) ↦ tv (struct ST, objectValues (*FM1* F ↦ tv (T, null) *FM2*))

requires $\neg_{Bool}$ pointer (S) in keys (ρ)
[transition]

RULE

k: * tv (T:*Type* *, member (S:*Expression*, F:*Id*)) ⇒ tv (T, null)

heap: pointer (S) ↦ tv (struct ST:*Id*, objectValues (—:*Map* F ↦ tv (T, undef) ⇒ tv (T, null) —:*Map*))

init-struct: PC ⇒ $PC \wedge_{Bool}$ pointer (S . F) == null

init-heap: pointer (S) ↦ tv (struct ST, objectValues (—:*Map* F ↦ tv (T, undef) ⇒ tv (T, null) —:*Map*))

[transition]

RULE

k: * tv (int * *, pointer (X:*Expression*)) ⇒ instanceNewSymbolicPointer (* X)

heap: pointer (X) ↦ tv (int *, undef) ⇒ tv (int *, pointer (* X))

init-struct: PC ⇒ $PC \wedge_{Bool}$ pointer (X) =/= null

init-heap: $\bullet_{Map}$ ⇒ pointer (X) ↦ tv (int *, pointer (* X))

[transition]

RULE

k: * tv (int * *, member (S:*Expression*, F:*Id*)) ⇒ instanceNewSymbolicPointer (S . F)

heap: pointer (S) ↦ tv (struct ST:*Id*, objectValues (*FM1*:*Map* F ↦ tv (int *, undef) ⇒ tv (int *, pointer (S . F)) *FM2*:*Map*))

init-struct: PC ⇒ $PC \wedge_{Bool}$ pointer (S . F) =/= null

init-heap: ρ:*Map* ⇒ pointer (S) ↦ tv (struct ST, objectValues (*FM1* F ↦ tv (int *, pointer (S . F)) *FM2*))

requires $\neg_{Bool}$ pointer (S) in keys (ρ)
[transition]

RULE

k

* tv (int * *, member (S:*Expression*, F:*Id*))
instanceNewSymbolicPointer (S . F)

heap

pointer (S) $\mapsto$ tv (struct ST:*Id*, objectValues (—:*Map* $F \mapsto$ tv (int *, undef) / tv (int *, pointer (S . F)) —:*Map*))

init-struct

PC
$PC \wedge_{Bool}$ pointer (S . F) =/= null

init-heap

pointer (S) $\mapsto$ tv (struct ST, objectValues (—:*Map* $F \mapsto$ tv (int *, undef) / tv (int *, pointer (S . F)) —:*Map*))

[transition]

RULE

k

* tv (struct S:*Id* * *, pointer (X:*Expression*))
initializeSymbolicObject (S, * X)

heap

pointer (X) $\mapsto$ tv (struct S *, undef) / tv (struct S *, pointer (* X))

init-struct

PC
$PC \wedge_{Bool}$ pointer (X) =/= null

init-heap

$\bullet_{Map}$
pointer (X) $\mapsto$ tv (struct S *, pointer (* X))

[transition]

RULE

k

* tv (struct S:*Id* * *, member (X:*Expression*, F:*Id*))
initializeSymbolicObject (S, X . F)

heap

pointer (X) $\mapsto$ tv (struct ST:*Id*, objectValues ($FM1$:*Map* $F \mapsto$ tv (struct S *, undef) / tv (struct S *, pointer (X . F)) $FM2$:*Map*))

init-struct

PC
$PC \wedge_{Bool}$ pointer (X . F) =/= null

init-heap

ρ:*Map*
pointer (X) $\mapsto$ tv (struct ST, objectValues ($FM1$ $F \mapsto$ tv (struct S *, pointer (X . F)) $FM2$))

requires $\neg_{Bool}$ pointer (X) in keys (ρ)
[transition]

RULE

k

* tv (struct S:*Id* * *, member (X:*Expression*, F:*Id*))
initializeSymbolicObject (S, X . F)

heap

pointer (X) $\mapsto$ tv (struct ST:*Id*, objectValues (—:*Map* $F \mapsto$ tv (struct S *, undef) / tv (struct S *, pointer (X . F)) —:*Map*))

init-struct

PC
$PC \wedge_{Bool}$ pointer (X . F) =/= null

init-heap

pointer (X) $\mapsto$ tv (struct ST, objectValues (—:*Map* $F \mapsto$ tv (struct S *, undef) / tv (struct S *, pointer (X . F)) —:*Map*))

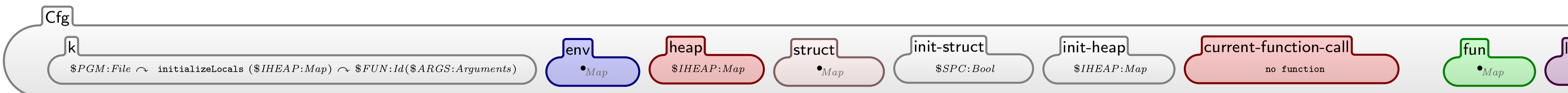
[transition]

END MODULE

MODULE KERNELC

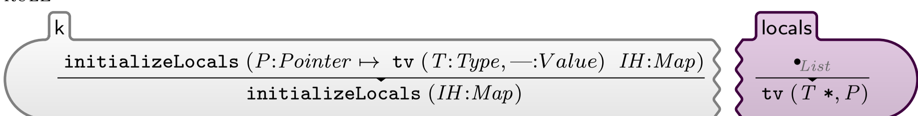
SYNTAX *KResult* ::= *EvaluatedExpression*

CONFIGURATION:



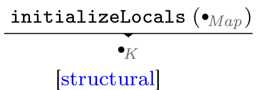
SYNTAX *Expression* ::= initializeLocals (*Map*)

RULE



[structural]

RULE



$$\frac{\texttt{initializeLocals}\;(\bullet_{Map})}{\bullet_{K}}$$

[structural]

END MODULE