$\mathbb{K}$ Definitions as Matching Logic Theories, Formally

Xiaohong Chen[1], Horaţiu Cheval[2,3], Dorel Lucanu[1,4], and Grigore Roşu[1,5]

[1] Pi Squared Inc. xiaohong.chen@pi2.network
[2] LOS Center, University of Bucharest, Romania horatiu.cheval@unibuc.ro
[3] Institute for Logic Data Science, Romania
[4] Alexandru Ioan Cuza University of Iaşi, Romania dorel.lucanu@info.uaic.ro
[5] University of Illinois at Urbana-Champaign, USA grosu@illinois.edu

Abstract. The $\mathbb{K}$ Framework is a state-of-the-art tool for designing and analyzing programming languages, relying on a frontend tool, kompile, to translate high-level $\mathbb{K}$ definitions into a low-level Kore representation based on Matching Logic ($\mathbb{ML}$). Currently, this compilation process lacks a formal specification, making it a trusted but unverified component in the toolchain. This paper addresses this gap by proposing a formal mechanism for obtaining the denotational semantics for $\mathbb{K}$ definitions directly as $\mathbb{ML}$ theories. A strong feature of the proposed approach is that it respects the abstraction and modularity principles of $\mathbb{K}$.

Keywords: $\mathbb{K}$ Framework · Matching Logic · Context Theory.

1 Introduction

The $\mathbb{K}$ Framework (https://kframework.org) allows for the design and modeling of programming languages and software/hardware systems. It is based on the programming, modeling, and specification language $\mathbb{K}$ [32]. The $\mathbb{K}$ Framework comes with tools for building interpreters, model checkers, verifiers, related documentation, and more. These tools are implemented generically once and for all and then instantiated by a language definition specified using $\mathbb{K}$.

The $\mathbb{K}$ process can be broadly separated into two phases: the *frontend phase* and the *backend phase* (see Figure 1, the white part). During the frontend stage, kompile compiles a $\mathbb{K}$ language definition into an intermediate representation known as the Kore format, used to specify matching logic ($\mathbb{ML}$) representations of the $\mathbb{K}$ definitions. It also infers the omitted parts of configurations in semantic rules and the types of all the variables. The tool kompile outputs a source file `definition.kore` that includes the entire matching logic theory encoding the formal semantics of the language. This Kore file is then passed to $\mathbb{K}$s backend to instantiate the corresponding language tools.

How kompile generates the $\mathbb{ML}$ encoding (in Kore format) of the formal language semantics is magic for $\mathbb{K}$ users. For example, the below $\mathbb{K}$ syntax declaration (taken from Figure 2)

N. Bertrand and S. Milius (Eds.): FoSSaCS 2026, LNCS 16503, pp. 198–219, 2026.
https://doi.org/10.1007/978-3-032-22730-0_10

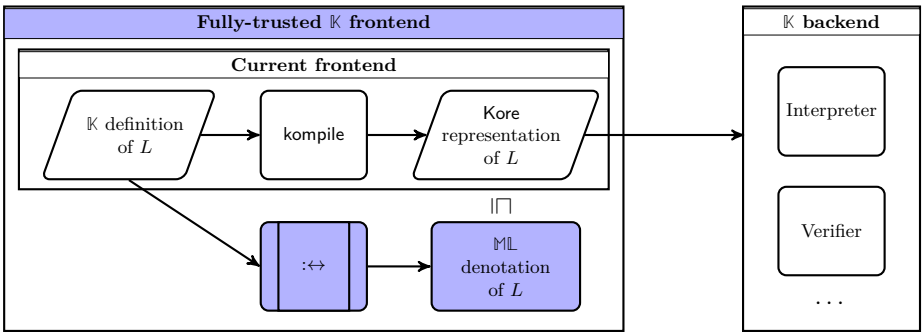

Fig. 1. 𝕂 process: the white boxes represent the current approach, the blue boxes the contribution of this paper. The relation :↔ represents a formal specification of kompile.

```
syntax Stmt ::= "inc" "(" Counter "," Label ")"
```

is compiled into a Kore[6] statement of the form

```
symbol Lblinc{}(SortCounter{}, SortLabel{}) : SortStmt{}
      [constructor{}(), functional{}(), injective{}]
```

together with the axioms corresponding to the attributes `constructor` (indicating that `Lblinc` is a constructor of `SortStmt`), `functional`, and `injective` (indicating that `Lblinc` is an injective function).

The kompile tool is fully developed and tested on many real-life programming languages (see Section 5). However, in order to completely trust that its output `definition.kore` represents indeed the matching logic denotation of the 𝕂 definition, we need a transparent formalization of this process.

The main contribution of this paper is to propose such a formalization, where the 𝕂 frontend definition can be seen as a *domain specific notation* of the 𝕄𝕃 theory it represents (blue part in Figure 1, where the notation relation is :↔). This formalization can be used further to show that kompile is correct by proving that the Kore representation it produces is a refinement (notation ⊑ in Figure 1) of the matching logic theory[7].

There are several significant challenges that must be addressed when formalizing 𝕂 definitions as matching logic theories, briefly explained below.

Axiomatization of abstract rules. 𝕂 allows to specify the rewrite rules describing the semantics in a very abstract way. A main advantage of this abstraction mechanism is modularity: if several languages share some functionality, then this functionality can be specified once and for all and imported by the corresponding language definitions. The tool kompile generates specific axioms for each language. The challenge is to have just one axiomatization for such rules to be shared by all language theories requiring it. The solution we propose is to use the 𝕄𝕃 *theory of contexts* (Sections 3 and 4.5).

[6] Kore uses a many-sorted version of matching logic [29,9].
[7] The correctness aspect is outside the scope of this paper.

Axiomatization of configurations. Configuration specification in $\mathbb{K}$ uses an XML-like notation, describing a tree structure of cells. The challenge is to find an appropriate axiomatic representation for such structures. Our solution consists in a specific abstract datatype theory for each kind of cells (Section 4.4).

Axiomatization of dynamic attributes. Abstraction is also obtained in $\mathbb{K}$ using dynamic attributes. For instance, arguments' order is specified by `strict`-like attributes. Since these attributes are themselves abstractions of certain semantic rules, their axiomatization is based on contexts as well (Sections 3 and 4.2).

Axiomatization of equality on sorts whose constructors satisfy some axioms. The $\mathbb{K}$ definition of a programming language can include datatypes whose constructors satisfy some axioms, such as associativity or commutativity. Hence, there are two kinds of equalities for these data structures: syntactic and semantic (modulo axioms). Our solution is based on using *quotient sorts* (Sections 2.2 and 4.3).

Structure of the paper Section 2 provides background on the two key formalisms. It first describes the components of a $\mathbb{K}$ definition, then introduces the functional variant of matching logic ($\mathbb{ML}$), covering relevant information on the theories used in the paper. Section 3 includes a central contribution: a formal theory of contexts within $\mathbb{ML}$. This reified specification is used to abstract away parts of a configuration, which is crucial for defining the denotations of $\mathbb{K}$'s dynamic features. Section 4 is the core technical contribution, explaining how to translate a $\mathbb{K}$ definition into an $\mathbb{ML}$ theory. We conclude with a discussion on related work (Section 5) and future directions (Section 6).

2 Preliminaries

2.1 $\mathbb{K}$ Frontend Definitions

In $\mathbb{K}$, a definition of a programming language consists of three main components:

Concrete syntax A conventional BNF grammar defining the structure of valid programs in the language. We distinguish three basic kinds of BNF productions:

- *sort declaration*: `syntax` ⟨*nonterminal*⟩;
- *subsorting*: `syntax` ⟨*nonterminal*⟩ ::= ⟨*nonterminal*⟩, and
- *symbol declaration*:
  `syntax` ⟨*nonterminal*⟩ ::= ⟨*list-nonterminals-and-terminals*⟩ [⟨*attributes*⟩].

A *function symbol* is a symbol declared with the attribute `function`. A non-function symbol is considered as being a *constructor* of the sort given by the left-hand side non-terminal.

Execution state (configuration) A representation of the state maintained by a program during execution. Its declaration is of the form

`configuration` ⟨*cfg*⟩

where ⟨*cfg*⟩ is a tree-like structure of cells, declared using an XML-like syntax.

Operational semantics A set of rewrite rules defining how the program's state changes over time. A rewrite rule

`rule` ⟨*rl*⟩ [`requires` ⟨*precond*⟩] [`ensures` ⟨*postcond*⟩]

can be a *simplification rule*, where ⟨*rl*⟩ is of the form ⟨*lhs*⟩ `=>` ⟨*rhs*⟩ and used to compute function symbols, or a *one-step transition* specification, where ⟨*rl*⟩ is of the form $C_1[\langle lhs_1 \rangle$ `=>` $\langle rhs_1 \rangle] \ldots C_k[\langle lhs_k \rangle$ `=>` $\langle rhs_k \rangle]$ and C_i specifies a sub-configuration where the local change ⟨lhs_i⟩ `=>` ⟨rhs_i⟩ holds.

𝕂 uses rewrite rules to define execution in a precise way. A rewrite rule takes the current configuration and transforms it into a new configuration based on a predefined mechanism, ensuring that the execution follows the formal semantics.

Running Example. A minimal computation model, Turing complete, and including all ingredients of a programming language, is the Minsky Machine [22]. It consists of two counters and the following statements: *inc(C, L)*—increments counter C and jumps to statement L; *decjz(C, L_1, L_2)*—if $C > 0$ decrements C and jumps to L_1, otherwise jumps to L_2; and *halt*—stops the execution.

A 𝕂 definition of the Minsky Machine is given in Figure 2. Module `MM-SYNTAX` includes the syntax for statements and counter names. Attribute `strict(1)` refers to the *strict* evaluation of the first argument (`Exp`), a.k.a. eager evaluation or call-by-value. This kind of evaluation is obtained through heating/cooling rules, which are automatically generated. Module `CONFIG` describes the configuration (execution state) consisting of three cells: `<k>`—for the sequential list of tasks to be executed, `<counters>`—for the two counters, and `<stmts>`—for the set of statements represented as pairs *label* ↦ *statement*. Module `MM-SEMANTICS` includes the rewrite rules describing the execution steps. The rule in line 22 is of the form $C_1[\langle lhs_1 \rangle$ `=>` $\langle rhs_1 \rangle]$ $C_2[\langle lhs_2 \rangle$ `=>` $\langle rhs_2 \rangle]$, where C_1 is the sub-configuration given by cell `<k>`, and C_2 is the sub-configuration given by cell `<stmts>`. The rule in line 27 is syntactic sugar for `<k>decjz(0, L1, _) => L1 ...</k>`. Each such rule represents an execution step *current-config* → *next-config* in an abstract way, using a configuration-abstraction/concretization mechanism. The rules in lines 30 and 31 are *simplification* rules because they are used to compute the function symbol `max`; they do not represent semantic execution steps, but function evaluations.

In Figure 3 we extend the 𝕂 definition of the Minsky Machine with multi-threading. Note that this extension is very modular: the syntax and semantics are augmented with new specific rules, and only the configuration is changed significantly. The multiplicity attribute in `<thread multiplicity = "*" ...>` says that cell `<thread>` can have zero, one, or more occurrences in a concrete configuration. *The configuration-abstraction/concretization mechanism allows to use the rewrite rules of the sequential machine in a multi-threading configuration.*

2.2 Matching Logic (𝕄𝕃)

In this paper, we use the functional variant of Matching Logic (𝕄𝕃) [6].

An 𝕄𝕃 signature (Σ, EV, SV) consists of a set of *constant symbols* Σ, a set of *element variables* EV, and a set of *set variables* SV. The formulas, called

```
module MM-SYNTAX
  imports INT
  syntax Counter ::= "c1" | "c2"
  syntax Label ::= Int
  syntax Exp ::= Counter | Int
  syntax Stmt ::= "inc" "(" Counter "," Label ")"
    | "decjz" "(" Exp "," Label "," Label ")" [strict(1)]
    | "halt"
  syntax Int ::= max(Int, Int) [function]
endmodule
module CONFIG
  imports MM-SYNTAX
  imports MAP
  configuration <T>
      <k> 0 </k>
      <counters> c1 |-> 0 c2 |-> 0 </counters>
      <stmts> $PGM:Map </stmts>
    </T>
endmodule
module MM-SEMANTICS
  imports CONFIG
  rule <k>L:Label => S</k> <stmts>...L |-> S:Stmt...</stmts>
  rule <k> inc(C:Counter, L:Label) => L ...</k>
       <counters>... C |-> (V => V +Int 1)  </counters>
  rule <k> C:Counter => max(0, V) ...</k>
       <counters>... C |-> (V => max(0,V -Int 1)) </counters>
  rule decjz(0, L1, _) => L1
  rule decjz(V, _, L2) => L2 requires V >Int 0
  rule halt => .K
  rule max(I1:Int, I2:Int) =>  I1 requires (I1 >Int I2)
  rule max(I1:Int, I2:Int) =>  I2 requires (I1 <=Int I2)
endmodule
```

Fig. 2. $\mathbb{K}$ definition of a Minsky Machine

patterns, are defined as follows:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\ \varphi_2 \mid \bot \mid \varphi_1 \to \varphi_2 \mid \exists x.\varphi \mid \mu X.\varphi \text{ if } \varphi \text{ is positive in } X$$

A pattern φ is positive in X if every free occurrence of X is under an even number of negations in φ, where a negation $\neg\psi$ is the notation $\psi \to \bot$ and subformulas $\psi_1 \to \psi_2$ are seen as $\neg\psi_1 \vee \psi_2$. Let PATTERN denote the set of patterns.

Definition 1 (Models). *Given $\mathbb{\Sigma} = (EV, SV, \Sigma)$, a* $\mathbb{\Sigma}$-model *(or simply* model*) is a tuple $(M, \_\bullet\_, \{M_\sigma\}_{\sigma\in\Sigma})$, where*

1. *M is a carrier set, required to be nonempty;*
2. *$\_\bullet\_ : M \times M \to \mathcal{P}(M)$ is a function, called the* interpretation of application*; here, $\mathcal{P}(M)$ is the powerset of M;*
3. *$M_\sigma \subseteq M$ is a subset of M, called the* interpretation of σ in M*, $\sigma \in \Sigma$.*

```
module MT-SYNTAX
  imports MM-SYNTAX
  syntax Stmt ::= "fork" "(" Label "," Label ")"
                | "join" "(" Counter "," Label ")"
                | "exit"
endmodule
module CONFIG
  imports MT-SYNTAX
  imports MAP
  configuration <T>
      <threads>
        <thread multiplicity = "*" type = "List">
          <k> 0 </k>
        </thread>
      </threads>
      <counters> c1 |-> 0 c2 |-> 0 </counters>
      <stmts> $PGM:Map </stmts>
    </T>
endmodule
module MT-SEMANTICS
  imports MM-SEMANTICS
  rule <threads>.Bag => <thread> <k>0</k> </thread></threads>
  rule <k> fork(L1, L2) => L2 ...</k>
       ( .Bag => <thread> <k> L1 </k> </thread> )
  rule <k> join(C, L) => L ...</k>
       <counters>... C |-> 0 </counters>
  rule <thread> <k> exit ...</k> </thread> => .Bag
endmodule
```

Fig. 3. 𝕂 definition of a Minsky Machine extended with multi-threading

By abuse of notation, we write M to denote the above model.

We further extend $\_\bullet\_$ *pointwise* from elements to sets as follows:

$$\_\bullet\_ : \mathcal{P}(M) \times \mathcal{P}(M) \to \mathcal{P}(M) \qquad A \bullet B = \textstyle\bigcup_{a\in A, b\in B} a \bullet b \;\text{ for } A, B \subseteq M$$

Note that $\emptyset \bullet A = A \bullet \emptyset = \emptyset$ for any $A \subseteq M$.

Definition 2 (Pattern interpretation). *Given $\Sigma = (EV, SV, \Sigma)$ and a model M, an* (M-)valuation *is a function $\rho\colon (EV \cup SV) \to (M \cup \mathcal{P}(M))$ that maps element variables to elements in M and set variables to subsets of M; i.e., $\rho(x) \in M$ for all $x \in EV$ and $\rho(X) \subseteq M$ for all $X \in SV$. We define a* pattern interpretation $|\_|_\rho\colon \textsc{Pattern} \to \mathcal{P}(M)$ *inductively as follows:*

$$|x|_\rho = \{\rho(x)\} \quad |X|_\rho = \rho(X) \quad |\sigma|_\rho = M_\sigma \quad |\bot|_\rho = \emptyset \quad |\varphi_1\,\varphi_2|_\rho = |\varphi_1|_\rho \bullet |\varphi_2|_\rho$$

$$|\varphi_1 \to \varphi_2|_\rho = M \setminus (|\varphi_1|_\rho \setminus |\varphi_2|_\rho) \qquad |\exists x.\,\varphi|_\rho = \bigcup_{a\in M} |\varphi|_{\rho[a/x]} \qquad |\mu X.\,\varphi|_\rho = \mu \mathcal{F}^\rho_{X,\varphi}$$

where $\rho[a/x]$ is the valuation ρ' such that $\rho'(x) = a$, $\rho'(y) = \rho(y)$ for any $y \in EV$ distinct from x, and $\rho'(X) = \rho(X)$ for any $X \in SV$. Here, $\mathcal{F}^\rho_{X,\varphi}\colon \mathcal{P}(M) \to$

$\mathcal{P}(M)$ is the function defined as $\mathcal{F}^{\rho}_{X,\varphi}(A) = |\varphi|_{\rho[A/X]}$ for every $A \subseteq M$, where $\rho[A/X]$ is the valuation ρ' such that $\rho'(X) = A$, $\rho'(Y) = \rho(Y)$ for any $Y \in SV$ distinct from X, and $\rho'(x) = \rho(x)$ for any $x \in EV$.

Definition 3. *1. φ is* valid in *M (equivalently, φ* holds in *M), written $M \models \varphi$, if $|\varphi|_\rho = M$, for any M-valuation ρ.*
2. If Γ is a set of patterns, then $M \models \Gamma$ if $M \models \varphi$ for any $\varphi \in \Gamma$.
3. $\Gamma \models \varphi$ if $M \models \varphi$ for any model M with $M \models \Gamma$. We say that φ is a semantic consequence *of Γ. If Γ is the empty set, we simply write $\models \varphi$.*

Notations [27] is a mechanism that allows flexible use of $\mathbb{ML}$. In particular, notations can be used to define a domain-specific logic inside $\mathbb{ML}$. It uses a less conventional notion of theory given as a triple $(\Sigma, \Phi, \vdash)$, where Σ is the alphabet of constant symbols, Φ a set of patterns, and $\vdash$ an entailment relation.

A *notation-based specification* is given as a conservative inclusion theory morphism $(\Sigma, \Phi, \vdash) \hookrightarrow (\Sigma, \Phi', \vdash')$, such that each *new formula* $\varphi' \in \Phi' \setminus \Phi$ is a *notation* of a formula $\varphi \in \Phi$, written as $\varphi' :\leftrightarrow \varphi$ and expressed by two new axioms: $\vdash' \varphi' \to \varphi$ and $\vdash' \varphi \to \varphi'$. Φ' might also contain additional axioms that limit the use of notations. By abuse of notation, we often write $(\Sigma, \Phi, \vdash) :\leftrightarrow (\Sigma, \Phi', \vdash')$ to express that $(\Sigma, \Phi', \vdash')$ is an extension by notation of $(\Sigma, \Phi, \vdash)$.

The simplest example is the definition of the derived operators as notations: $\neg\varphi :\leftrightarrow \varphi \to \bot$, $\varphi_1 \vee \varphi_2 :\leftrightarrow \neg\varphi_1 \to \varphi_2$, $\top :\leftrightarrow \neg\bot$, etc.

A Hilbert-style proof system exists for $\mathbb{ML}$ [9,6], establishing the provability relation $\Gamma \vdash \varphi$, where Γ is a matching logic theory and φ is a pattern. This proof system is very expressive as a variety of common logics can be derived from it. Therefore, we believe that it is suitable for deriving specific proof rules for $\mathbb{K}$ definitions. However, familiarity with this proof system is not required to understand the technical results presented in this paper.

2.3 Basic $\mathbb{ML}$ Theories

The $\mathbb{ML}$ denotation of a $\mathbb{K}$ definition is built on top of several theories.

Theory of equality It includes a symbol def axiomatically defined by the axiom $\forall x.\, \mathsf{def}\, x$ and having the following meaning: $|\mathsf{def}\, \varphi|_{M,\rho} = \textbf{if } |\varphi|_{M,\rho} \neq \emptyset \textbf{ then } M \textbf{ else } \emptyset$. Totality, equality, inclusion, and membership are specified as notations: $\lceil\varphi\rceil :\leftrightarrow \mathsf{def}\, \varphi$ (definedness), $\lfloor\varphi\rfloor :\leftrightarrow \neg\lceil\neg\varphi\rceil$ (totality), $\varphi_1 = \varphi_2 :\leftrightarrow \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor$ (equality), $\varphi_1 \subseteq \varphi_2 :\leftrightarrow \lfloor\varphi_1 \to \varphi_2\rfloor$ (set inclusion), and $x \in \varphi :\leftrightarrow x \subseteq \varphi$ (membership). Using equality, e.g., we may define that a pattern φ is *functional* if axiom $\exists x.\varphi = x$ holds, i.e., it is always evaluated to singleton. An example of functional patterns is given by the (algebraic) terms.

Theory of sorts A symbol inh is used such that the pattern $(\mathsf{inh}\ s)$ represents the set of all inhabitants of the sort with name s. Notation $\top_s :\leftrightarrow (\mathsf{inh}\ s)$ is often used for the pattern representing the set of inhabitants. Symbol Sort is used to represent the sort of all sort names, i.e., its inhabitants are the sort names; in particular, $\mathsf{Sort} \in \top_{\mathsf{Sort}}$.

$\mathbb{ML}$ allows to specify operations over sorts [6,8]. Next, we present only some of them.

Product sorts. Given two sorts s_1 and s_2 (i.e., $s_1, s_2 \in \top_{\mathsf{Sort}}$), their *product sort* is specified by a sort name $s_1 \otimes s_2$, $s_1 \otimes s_2 \in \top_{\mathsf{Sort}}$, a constant symbol $\langle \_, \_ \rangle$ in Σ, together with a notation $\langle x, y \rangle :\leftrightarrow \langle \_, \_ \rangle\ x\ y$ and the following axioms:

$$\forall x{:}s_1.\forall y{:}s_2.\exists z{:}s_1 \otimes s_2.\langle x, y \rangle = z \qquad \langle x_1, y_1 \rangle = \langle x_2, y_2 \rangle \to x_1 = x_2 \wedge y_1 = y_2$$
$$\top_{s_1 \otimes s_2} = \exists x{:}s_1.\exists y{:}s_2.\langle x, y \rangle \qquad \forall x{:}s_1.\forall y{:}s_2.\forall z{:}s_3.\langle \langle x, y \rangle, z \rangle = \langle x, \langle y, z \rangle \rangle$$

Due to associativity, $\langle x, y, z \rangle$ equally denotes either $\langle \langle x, y \rangle, z \rangle$ or $\langle x, \langle y, z \rangle \rangle$. This naturally extends to tuples of arbitrary lengths: $\langle x_1, \ldots, x_k \rangle$, $k \geq 2$.

Remark 1. If $\varphi{:}s_1$ and $\psi{:}s_2$ are two (nonfunctional) patterns, then $\langle \varphi, \psi \rangle$ denotes the cartesian product of φ and ψ, as

$$|\langle \varphi, \psi \rangle|_\rho = \textstyle\bigcup_{a \in |\varphi|_\rho, b \in |\psi|_\rho} (a, b).$$

Power sorts. Given a sort s ($s \in \top_{\mathsf{Sort}}$), its *power sort* is specified by a sort name 2^s, $2^s \in \top_{\mathsf{Sort}}$, two constant symbols, $\mathsf{extension}$ and $\mathsf{intension}$ in Σ, together with the following axioms:

$$\forall \alpha{:}2^s.\mathsf{extension}\ \alpha \subseteq \top_s \qquad \forall \alpha{:}2^s.\forall \beta{:}2^s.\mathsf{extension}\ \alpha = \mathsf{extension}\ \beta \to \alpha{=}\beta$$
$$X \subseteq \top_s \to \exists \alpha{:}2^s.\mathsf{extension}\ \alpha{=}X \qquad \mathsf{intension}\ \varphi :\leftrightarrow \exists \alpha{:}2^s.\alpha \wedge (\mathsf{extension}\ \alpha = \varphi)$$

A *relation* $R \subseteq \top_{s_1} \times \top_{s_2}$ is presented by an element $r : 2^{s_1 \otimes s_2}$ such that $\mathsf{extension}\ r = R$. The *inverse* R^{-1} of a relation $R \subseteq \top_s \times \top_s$ is specified by a constant symbol $\_^{-1}$, a notation, and two axioms:

$$r^{-1} :\leftrightarrow \_^{-1}\ r \qquad \forall r : 2^{s \otimes s}.\exists r' : 2^{s \otimes s}.r^{-1} = r'$$
$$\forall r : 2^{s \otimes s}.\mathsf{extension}\ r^{-1} = \exists x, y{:}s.\langle y, x \rangle \wedge \langle x, y \rangle \in \mathsf{extension}\ r$$

The *composition* $R_1 \circ R_2$ of two relations $R_1 \subseteq \top_{s_1} \times \top_{s_2}$ and $R_2 \subseteq \top_{s_2} \times \top_{s_3}$ is specified by a constant symbol $\_\circ\_$, a notation, and two axioms:

$$r_1 \circ r_2 :\leftrightarrow \_\circ\_\ r_1\ r_2 \qquad \forall r_1 : 2^{s_1 \otimes s_2}.\forall r_2 : 2^{s_2 \otimes s_3}.\exists r : 2^{s_1 \otimes s_3}.r = r_1 \circ r_2$$
$$\forall r_1 : 2^{s_1 \otimes s_2}.\forall r_2 : 2^{s_2 \otimes s_3}.\mathsf{extension}\ r_1 \circ r_2 = \exists x{:}s_1.\exists y{:}s_2.\exists z{:}s_3.$$
$$\langle x, z \rangle \wedge \langle x, y \rangle \in \mathsf{extension}\ r_1 \wedge \langle y, z \rangle \in \mathsf{extension}\ r_2$$

Function Sorts. Given sorts s_1, s_2 ($s_1, s_2 \in \top_{\mathsf{Sort}}$), the functions $\top_{s_1} \to \top_{s_2}$ are specified by a *function sort* $[s_1 \to s_2]$, where the functions are represented by their graphs, i.e., $\top_{[s_1 \to s_2]} \subseteq \top_{2^{s_1 \otimes s_2}}$, and the following domain-specific axioms:

$$\forall f{:}[s_1 \to s_2].\forall x{:}s_1.\exists y{:}s_2.\langle x, y \rangle \in \mathsf{extension}\ f$$
$$\forall f{:}[s_1 \to s_2].\forall x{:}s_1.\forall y_1, y_2{:}s_2.$$
$$(\langle x, y_1 \rangle \in \mathsf{extension}\ f \wedge \langle x, y_2 \rangle \in \mathsf{extension}\ f) \to y_1 = y_2$$
$$\forall f{:}[s_1 \to s_2].\forall x{:}s_1.f\ x = \exists y.y \wedge \langle x, y \rangle \in \mathsf{extension}\ f$$

Quotient Sorts. Given a sort s ($s \in \top_{\mathsf{Sort}}$) and an equivalence relation $\equiv$ on $\top_s \times \top_s$, the *quotient sort* is specified by a sort name $s/{\equiv}$ ($s/{\equiv} \in \top_{\mathsf{Sort}}$), a constant symbol $[\_]_\equiv$ together with a notation $[x]_\equiv :\leftrightarrow [\_]_\equiv\ x$, and the axioms:

$$\forall x{:}s.[x]_\equiv \in \top_{2^s} \quad \top_{s/\equiv} = \exists x{:}s.[x]_\equiv \qquad \forall x{:}s.\mathsf{extension}\ [x]_\equiv = \exists y{:}s.y \wedge x \equiv y$$

The pattern $[x]_\equiv$ uniquely identifies the equivalence class of x, represented by

extension $[x]_{\equiv}$: if $x \equiv y$ then $[x]_{\equiv} = [y]_{\equiv}$. The following hold:

$\forall f : [s \to s'].(\forall x, y{:}s.x \equiv y \to (f\ x = f\ y)) \to \exists \bar{f}{:}[(s/_{\equiv}) \to s'].(\forall x{:}s.\bar{f}\ [x]_{\equiv} = f\ x)$

$\forall \bar{f} : [(s/_{\equiv}) \to s'].\exists f{:}[s \to s'].(\forall x{:}s.\bar{f}\ [x]_{\equiv} = f\ x)$

Specification of rewriting The one-step rewriting (transition) relation is specified using a *one-path next* symbol $\bullet$ having the following interpretation: for each $x \in \bullet\varphi'$ there is a $y \in \varphi'$ such that $x \Rightarrow y$ is a one-step rewriting relation. Patterns φ' and $\bullet\varphi'$ must have the same sort: $\varphi' \subseteq \top_s \to \bullet\varphi' \subseteq \top_s$. Furthermore, we assume that $\bullet\varphi'$ is defined pointwise: $\bullet\varphi' = \exists y{:}s.\ \bullet y \wedge y \in \varphi'$. A rewrite rule, specifying a particular one-step rewriting relation, is a pattern of the form $\varphi \to \bullet\varphi'$. In order to emphasize that $\varphi \to \bullet\varphi'$ indeed represents a relation, we introduce the following notation:

$$\mathsf{CH}(\varphi \to \bullet\varphi') :\leftrightarrow \exists x{:}s.\exists y{:}s.\langle x, y\rangle \wedge x \in (\varphi \wedge \bullet y) \wedge y \in \varphi'$$

where a rewriting step $x \Rightarrow y$ can be seen as a notation for the pair $\langle x, y\rangle$ in the above pattern. This can be interpreted as a "particular case" of the CurryHoward correspondence, where the implication corresponds to a relation (instead of a function). Note that $\mathsf{CH}(\varphi \to \bullet\varphi')$ is a pattern of sort $s \otimes s$. In particular, when φ is $\top_s$, we have $\mathsf{CH}(\bullet\varphi') :\leftrightarrow \exists x{:}s.\exists y{:}s.\langle x, y\rangle \wedge x \in \bullet y \wedge y \in \varphi'$.

Remark 2. If a rewrite pattern $\varphi \Rightarrow \varphi' :\leftrightarrow \varphi \to \bullet\varphi'$ is an axiom of a theory Γ, then we have $M \models \varphi \to \bullet\varphi'$ iff $M \models \varphi \subseteq \bullet\varphi'$ for any Γ-model M.

3 A Theory of Contexts

A *context* is intended to represent a pattern φ with a hole, usually denoted by $\Box$, which can be filled by some other pattern ψ. Although it would be straightforward to define contexts as meta-theoretical constructs, e.g., operations on the syntax, such an approach would not be aligned with our goal of formalizing the semantics of $\mathbb{K}$ as an $\mathbb{ML}$ theory. We need a reified specification, given through a signature and some axioms, together with notations on top of it. For this, we apply the general mechanism for defining binders introduced in [8], which starts with the following notation

$$[\Box{:}s_1]\varphi :\leftrightarrow \mathsf{intension}\ (\exists \Box{:}s_1.\langle \Box, \varphi\rangle)$$

where $\Box$ is an element variable of sort s_1 and φ is a functional pattern of sort s_2. The $[\Box{:}s_1]\varphi$ notation, of sort $2^{s_1 \otimes s_2}$, is used to indicate that variable $\Box$ is bound in φ, and is the building block for encoding the binders in $\mathbb{ML}$.

The extension of a binding is a functional relation:

$$\begin{aligned} \mathsf{extension}\ [\Box{:}s_1]\varphi &:\leftrightarrow \mathsf{extension}\ (\mathsf{intension}\ (\exists \Box{:}s_1.\langle \Box, \varphi\rangle)) && \text{(by notation)} \\ &= \exists \Box{:}s_1.\langle \Box, \varphi\rangle && \text{(by power sort def.)} \end{aligned}$$

If ρ is a variable valuation, then $|\exists \Box{:}s_1.\ \langle \Box, \varphi\rangle|_\rho = \bigcup_{a \in M_{s_1}} |\langle \Box, \varphi\rangle|_{\rho[a/\Box]} = \bigcup_{a \in M_{s_1}} \langle a, |\varphi|_{\rho[a/\Box]}\rangle$, where $M_{s_1} = |\top_{s_1}|_\rho$. Since φ is functional, it follows that the relation is the graph of a function.

Starting with the theory of sorts from Section 2.2, we add the symbols

$$\mathsf{plug},\ \mathsf{gamma},\ \text{and}\ \mathsf{Context}^{s_2}_{s_1}\ \text{for any sorts}\ s_1, s_2.$$

$\mathsf{Context}^{s_2}_{s_1}$ stands for the sort of contexts of sort s_2 with a hole of sort s_1. Constant symbol gamma defines a binder used to specify that a certain element variable $\Box$ is treated as a hole in a pattern, thus forming a context, while plug is used to substitute the bound hole for another pattern. We use the following notations:

$$\gamma\Box{:}s_1.\varphi :\leftrightarrow \mathsf{gamma}([\Box{:}s_1]\varphi) \qquad C[x] :\leftrightarrow \mathsf{plug}(C, x)$$

The specification $\Gamma_{\mathsf{context}}$ of the contexts is given by the following axioms:

$$\forall s_1, s_2{:}\mathsf{Sort}.\mathsf{Context}^{s_2}_{s_1} \in \top_{\mathsf{Sort}} \tag{Ax.1}$$

$$\forall s_1, s_2{:}\mathsf{Sort}.\forall \alpha_1, \alpha_2{:}[s_1 \to s_2].(\mathsf{gamma}\ \alpha_1 = \mathsf{gamma}\ \alpha_2) \to \alpha_1 = \alpha_2 \tag{Ax.2}$$

$$\forall s_1, s_2{:}\mathsf{Sort}.\top_{\mathsf{Context}^{s_2}_{s_1}} = \exists \alpha{:}[s_1 \to s_2].\mathsf{gamma}\ \alpha \tag{Ax.3}$$

$$\forall s_1, s_2{:}\mathsf{Sort}.\forall C_1, C_2{:}\mathsf{Context}^{s_2}_{s_1}.(\forall x{:}s_1.C_1[x] = C_2[x]) \to C_1 = C_2 \tag{Ax.4}$$

$$\forall s_1, s_2{:}\mathsf{Sort}.\forall C{:}\mathsf{Context}^{s_2}_{s_1}.\forall x{:}s_1.$$
$$C[x] = \exists \alpha{:}[s_1 \to s_2].\exists y{:}s_2.y \wedge (C = \mathsf{gamma}\ \alpha \wedge \langle x, y\rangle \in \mathsf{extension}\ \alpha) \tag{Ax.5}$$

Most of the axioms are self-explanatory, considering the meaning of gamma and plug. The only rather technical axiom is Ax.5, which, for $C :\leftrightarrow \gamma\Box{:}s_1.\varphi$, allows us to compute the plugging $C[z]$ in terms of φ:

$$C[z] = \exists y{:}s_2.y \wedge (\langle z, y\rangle \in \exists \Box{:}s_1.\langle \Box, \varphi\rangle).$$

By Axiom Ax.3, each context C is of the form $\mathsf{gamma}\ \alpha$. We may obtain $C :\leftrightarrow \gamma\Box{:}s_1.\varphi$, where $\varphi :\leftrightarrow \alpha(\Box)$ and $\alpha(\Box) :\leftrightarrow \exists y.y \wedge \langle \Box, y\rangle \in \mathsf{extension}\ \alpha$. Since α is a function, it follows that y is uniquely determined. Therefore, we always consider the contexts to be of the form $\gamma\Box{:}s_1.\varphi$, with φ functional.

The term 'context' is often used for constructs supporting a plugging operation that can be performed without constraints on capture avoidance. By contrast, in our formulation, plugging behaves like a capture avoiding substitution. While this is not strictly necessary, it is a consequence of our choice to internalize their definition as a theory, rather than presenting them as syntactical objects. This restriction helps avoid soundness issues related to accidental variable capture, and it does not affect the use of contexts for 𝕂 definitions.

Crucially, from these axioms we are able to prove that the plugging operation corresponds to substitution in the following sense, which is equivalent to saying that β-reduction relative to plug holds.

Theorem 1. *Let s_1, s_2 be two sorts, and φ be a pattern of sort s_2. Then,*

$$\Gamma_{\mathit{context}} \models (\gamma\Box{:}s_1.\varphi)[\psi] = \varphi[\psi/\Box]$$

for any functional pattern ψ of sort s_1 not having $\Box$ as a free element variable.

Given $C_1{:}\mathsf{Context}^{s_2}_{s_1}$ and $C_2{:}\mathsf{Context}^{s_3}_{s_2}$, the composition of C_1 and C_2 is defined as the following notation of sort $\mathsf{Context}^{s_3}_{s_1}$, and behaves similarly to function composition:

$$C_2 \circ C_1 :\leftrightarrow \gamma\Box{:}s_1.C_2[C_1[\Box]] \qquad (C_2 \circ C_1)[\psi] = C_2[C_1[\psi]]$$

Because $\mathbb{K}$ allows multiple rewrites to appear in the same rule (as long as they are not nested), we will need a notion of *multihole context* to formalize this feature, which is straightforward to define via currying.

An n-hole context on sorts $s_1, \ldots, s_n$ is a pattern of the form

$$C :\leftrightarrow \gamma\Box_1{:}s_1.\ldots.\gamma\Box_n{:}s_n.\varphi$$

where φ is of sort s. We will abbreviate the sort of such patterns by $\mathsf{Context}^{s}_{s_1,\ldots,s_n}$. As in currying, plugging into multihole contexts is simply iterated single-hole plugging: $C[\psi_1, \ldots, \psi_n] :\leftrightarrow C[\psi_1]\ldots[\psi_n]$.

The composition is naturally extended to contexts with multiple holes.

Definition 4 (Multiple context composition). *Let $C{:}\mathsf{Context}^{s}_{s_1,\ldots,s_n}$ be an n-hole context, and $C_i{:}\mathsf{Context}^{s_i}_{s'_i}$ with $i = 1, \ldots, n$ be contexts. We define the* multiple composition *of C and $C_1, \ldots, C_n$ as*

$$C \odot \langle C_1, \ldots, C_n\rangle :\leftrightarrow \gamma\Box_1{:}s'_1 \ldots \gamma\Box_n{:}s'_n.C[C_1[\Box_1], \ldots, C_n[\Box_n]],$$

where $\Box_1, \ldots, \Box_n$ are fresh element variables.

For $n = 1$, the multiple composition becomes *functional composition*: $C \circ C_1 \leftrightarrow C \odot C_1$. Note that $C \odot \langle C_1, \ldots, C_n\rangle$ will be of sort $\mathsf{Context}^{s}_{s'_1,\ldots,s'_n}$. The following result shows how to plug a multiple context composition.

Proposition 1 (Plugging multiple context composition). *Consider the contexts $C{:}\mathsf{Context}^{s}_{s_1,\ldots,s_n}$ and $C_i{:}\mathsf{Context}^{s_i}_{s'_i}$, $i = 1, \ldots, n$. For any functional patterns $\psi_i{:}s'_i$ with $i = 1, \ldots, n$, we have that*

$$\Gamma_{\mathsf{context}} \models (C \odot \langle C_1, \ldots, C_n\rangle)[\psi_1, \ldots, \psi_n] = C[C_1[\psi_1], \ldots, C_n[\psi_n]].$$

In this section, we focused on the aspects of the theory of contexts that are strictly required for the $\mathbb{ML}$ denotations in Section 4. A more comprehensive presentation can be found in [3].

4 $\mathbb{K}$ Definitions as $\mathbb{ML}$ Theories

In this section, we formally specify a notation $:\leftrightarrow$ such that

$$\mathbb{K} \text{ definition of } L \ :\leftrightarrow \ L^{\mathbb{ML}}$$

where $L^{\mathbb{ML}}$ is the matching logic theory represented by L.

4.1 The Main Idea

The main idea to generate the theory $L^{\mathbb{ML}}$ is as follows:

1. Let Sort be the constant symbol whose inhabitants are given by the set of sorts used in $L^{\mathbb{ML}}$.
2. Each non-terminal N denotes a sort N subsorted to KItem (see Section 4.3):
$$\texttt{syntax } N \ :\leftrightarrow \ \begin{array}{l}\exists y{:}\mathsf{Sort}.N = y\\ \top_N \subseteq \top_{\mathsf{KItem}}\end{array}$$
3. Each subsorting BNF production denotes a subsort axiom:

$\texttt{syntax}\ A\ \texttt{::=}\ B \ :\leftrightarrow\ \top_B \subseteq \top_A$

We also use notation $B <: S$ for the subsort relation.

4. Each non-subsorting BNF production denotes a distinct symbol $\sigma \in \Sigma(L^{\mathbb{ML}})$, which is functional:

$$\texttt{syntax}\ N\ \texttt{::=}\ T_1\ N_1 \ldots T_k\ N_k\ T_{k+1} \ :\leftrightarrow\ \begin{array}{l}\forall x_1{:}N_1 \ldots \forall x_k{:}N_k.\exists y{:}N. \\ \sigma(x_1, \ldots, x_k) = y\end{array}$$

In this way, each 𝕂 term t denotes, via the notation, an 𝕄𝕃 pattern $t^{\mathbb{ML}}$, i.e., $t \ :\leftrightarrow\ t^{\mathbb{ML}}$

5. Each attribute of a BNF production, excepting `symbol` and `function`, denotes a set of axioms (see Section 4.2). So, the syntax $BNF(L)$ of L denotes an 𝕄𝕃 theory $BNF(L)^{\mathbb{ML}}$ [5]:

$$BNF(L) \ :\leftrightarrow\ BNF(L)^{\mathbb{ML}}$$

6. The cells of the configuration denote a theory including specific sets of sorts, symbols, and axioms (see Section 4.4). So, the configuration declaration of L denotes an 𝕄𝕃 theory $CONFIG(L)^{\mathbb{ML}}$:

$$CONFIG(L) \ :\leftrightarrow\ CONFIG(L)^{\mathbb{ML}}$$

7. The 𝕂 simplification rewrite rules, which compute the value of a function symbol, denote equalities of the corresponding 𝕄𝕃 patterns.
8. The 𝕂 semantic rewrite rules denote one-step transition 𝕄𝕃 patterns, built using contexts (see Section 4.5). In this way, the semantic rewrite rules of L, $SEM(L)$, denote an 𝕄𝕃 theory $SEM(L)^{\mathbb{ML}}$:

$$SEM(L) \ :\leftrightarrow\ SEM(L)^{\mathbb{ML}}$$

9. The builtin modules are specified as abstract datatypes. An example is given in Section 4.3. Let $BLTN(L)^{\mathbb{ML}}$ denote the theory including the specifications of the builtin datatypes.
10. The 𝕄𝕃 theory denoted by L will be $L^{\mathbb{ML}} = BNF(L)^{\mathbb{ML}} \cup CONFIG(L)^{\mathbb{ML}} \cup SEM(L)^{\mathbb{ML}} \cup BLTN(L)^{\mathbb{ML}}$ and satisfies $L \ :\leftrightarrow\ L^{\mathbb{ML}}$.

4.2 Attributes of Symbols

Each BNF production defining a symbol may have, implicitly or explicitly, one or more attributes, grouped as follows:

- *definitional attributes*: `symbol`, `function`;
- *static semantic attributes*: `constructor`, `functional`, `total`, `injective`, `assoc`, `comm`, `unit`, `idem`;
- *dynamic semantic attributes*: `strict`, `seqstrict`, `strict`$(i_1, \ldots, i_k)$.

We use $\pi_\sigma(N_1, \ldots, N_k, N)$ to denote a non-subsorting BNF production for the non-terminal N with the attribute `symbol`(σ), where $N_1, \ldots, N_k$ are the non-terminals occurring in the BNF expression (in this order).

Definitional attributes The attribute `symbol`, designates the symbol name associated with that rule. It is mentioned implicitly or explicitly. The attribute `function` specifies that the symbol associated to that rule denotes a function.

Such symbols should have associated (simplification) rules that compute the value of the function, denoting equalities of the corresponding $\mathbb{ML}$ patterns:

$$\texttt{rule}\ t \Rightarrow t'\ \texttt{[simplification]} \ :\leftrightarrow\ t^{\mathbb{ML}} = t'^{\mathbb{ML}}$$

Example 1. Since the BNF production for `max` in Figure 2 has the attribute `function`, the rules computing it are simplification rules and their $\mathbb{ML}$ denotations are equalities. For example,

$$\texttt{rule max(I1:Int, I2:Int) => I1 requires (I1 >Int I2)}$$
$$:\leftrightarrow$$
$$\forall i_1, i_2{:}\mathsf{Int}.\ >_{\mathsf{Int}} (i_1, i_2) \rightarrow \mathsf{max}(i_1, i_2) = i_1$$

Static Semantic Attributes We present only the axioms denoted by the attribute `constructor`. The $\mathbb{ML}$ denotations of the other static semantic attributes are intuitive and therefore they are omitted.

1. For each $\pi_\sigma(N_1, \ldots, N_k, N)$ having the attribute `constructor`, an axiom saying that σ is injective is added, which is equivalent to the slogan "no confusion, the same constructor":
$$\forall x_1, x_1'{:}N_1 \ldots \forall x_k, x_k'{:}N_k.(\sigma(x_1, \ldots, x_k) = \sigma(x_1', \ldots, x_k')) \rightarrow (x_1 = x_1' \wedge \cdots \wedge x_k = x_k')$$
2. For each pair $\pi_\sigma(N_1, \ldots, N_k, N)$ and $\pi_{\sigma'}(N_1', \ldots, N_{k'}', N)$ with $\sigma \neq \sigma'$, having the attribute `constructor`, an axiom formalizing the slogan "no confusion, different constructors" is added:
$$\forall x_1{:}N_1 \ldots \forall x_k{:}N_k.\forall x_1'{:}N_1' \ldots \forall x_{k'}'{:}N_{k'}'.\sigma(x_1, \ldots, x_k) \neq \sigma'(x_1', \ldots, x_{k'}'))$$
3. If $\pi_{\sigma^i}(N_1^i, \ldots, N_{k_i}^i, N)$, $i \in I$, are all the productions having (explicitly or implicitly) the `constructor` attribute for the sort N, then the following axiom specifies the carrier set of N, formalizing the slogan "no junk":
$$\top_N = \mu X.(\bigvee_{s <: N} \top_s) \vee (\bigvee_{i \in I} \sigma^i(Y_1, \ldots, Y_{k_i})) \qquad \text{(No-Junk)}$$
where $Y_j = X$ if $N_j^i = N$, and $Y_j = \top_{N_j^i}$ if $N_j^i \neq N$.

Remark 3. 1. We assume that all sorts N are non-void: $\lceil \top_N \rceil$.
2. If the BNF productions are mutually recursive, then computing the carrier set using (No-Junk) is a bit tricky. For such cases, the technique described by Chen et al. [5] can be applied.

If some constructors of sort N have attributes `assoc`, `comm`, `idem`, and/or `unit`, then axioms defining the quotient sort $N/\cong_N$ are added, where $\cong_N$ is the congruence generated by these axioms. An example is given in Section 4.3.

Dynamic Semantic Attributes We consider only the case of the attribute `strict`$(i_1, \ldots, i_k)$ for $\pi_\sigma(N_1, \ldots, N_n, N)$, the other ones being particular cases

(syntactic sugar) of this. In 𝕂, sort KResult is used to specify the "values" of the programming language. We use notation $\mathsf{KResult}(x) :\leftrightarrow x \in \top_{\mathsf{KResult}}$ to say that x is a value already evaluated.

The expressions to be evaluated always lie in the `<k>` cells, therefore we have to use the contexts in order to abstract the configurations over which the computations are executed. The content of a cell `<k>` is a sequential list $t_1 \curvearrowright t_2 \curvearrowright \ldots$, meaning that t_1 is evaluated/computed first, then t_2, and so on.

For each $i \in \{i_1, \ldots, i_k\}$ of `strict` we have a *heating* rewriting axiom:

$$\forall C : \mathsf{Context}^{\mathsf{Cell}\langle\mathsf{T}\rangle}_{\mathsf{Cell}\langle\mathsf{k}\rangle}.\forall\kappa{:}\mathsf{K}.\forall x_1{:}N_1, \ldots, x_n{:}N_n.$$
$$(C \circ C_{\mathtt{k}})[\sigma(x_1, \ldots, x_n)] \wedge \neg\mathsf{KResult}(x_i) \rightarrow (\bullet C \circ C_{\mathtt{k}})[x_i \curvearrowright C_{\sigma,i}]$$

and a *cooling* rewriting axiom:

$$\forall C : \mathsf{Context}^{\mathsf{Cell}\langle\mathsf{T}\rangle}_{\mathsf{Cell}\langle\mathsf{k}\rangle}.\forall\kappa{:}\mathsf{K}.\forall x_1{:}N_1, \ldots, x_n{:}N_n.$$
$$(C \circ C_{\mathtt{k}})[x_i \curvearrowright C_{\sigma,i}] \wedge \mathsf{KResult}(x_i) \rightarrow \bullet(C \circ C_{\mathtt{k}})[C_{\sigma,i}[x_i]]$$

where $C_{\mathtt{k}} :\leftrightarrow \gamma\Box_{\mathtt{k}}{:}\mathsf{KItem}.\langle\!|\mathsf{k}|\!\rangle(\Box_{\mathtt{k}} \curvearrowright \kappa)$ and $C_{\sigma,i} :\leftrightarrow \gamma\Box_i{:}N_i.\sigma(x_1, \ldots, \Box_i, \ldots, x_n)$. Sorts K and KItem are described in Section 4.3 and $\mathsf{Cell}\langle\_\rangle$ in Section 4.4.

The heating rules correspond to the "split" operation, where a term is split into a context and a subterm (to be evaluated), while the cooling rules correspond to the "plug" operation, where the value obtained by the evaluation of the subterm is plugged into the context.

Example 2. Here are the axioms corresponding to `strict(1)` on `decjz` in the syntax of the Minsky Machine (see Figure 2):

$$\forall C{:}\mathsf{Context}^{\mathsf{Cell}\langle\mathsf{T}\rangle}_{\mathsf{Cell}\langle\mathsf{k}\rangle}.\forall\kappa{:}\mathsf{K}.\forall x_1{:}\mathsf{Exp}.\forall x_2, x_3{:}\mathsf{Label}.$$
$$(C \circ C_{\mathtt{k}})[\mathsf{decjz}(x_1, x_2, x_3)] \wedge \neg\mathsf{KResult}(x_1) \rightarrow \bullet(C \circ C_{\kappa})[x_1 \curvearrowright C_{\mathsf{decjz},1}]$$
$$\forall C{:}\mathsf{Context}^{\mathsf{Cell}\langle\mathsf{T}\rangle}_{\mathsf{Cell}\langle\mathsf{k}\rangle}.\forall\kappa{:}\mathsf{K}.\forall x_1{:}\mathsf{Exp}.\forall x_2, x_3{:}\mathsf{Label}.$$
$$(C \circ C_{\mathtt{k}})[x_1 \curvearrowright C_{\mathsf{decjz},1}] \wedge \mathsf{KResult}(x_1) \rightarrow \bullet(C \circ C_{\mathtt{k}})[C_{\mathsf{decjz},1}[x_1]]$$

where $C_{\mathsf{decjz},1}$ is the context $\gamma\Box{:}\mathsf{Exp.decjz}(\Box, x_2, x_3) \in \mathsf{Context}^{\mathsf{Stmt}}_{\mathsf{Exp}}$. These axioms can be instantiated in both configurations defined in Figures 2 and 3.

4.3 Builtin Theories

𝕂 includes a set of builtin datatypes [33], whose operations are implemented using a *hooking* mechanism. The datatypes are also specified in 𝕂 [33], so we can use these specifications to define their 𝕄𝕃 denotations. Here we consider the K datatype, which is an important component used in almost any 𝕂 definition.

K Datatype The K datatype is an associative list $t_1 \curvearrowright t_2 \curvearrowright \ldots$ [31] and is used to populate the cells `<k>`. Its 𝕂 specification and 𝕄𝕃 denotation are given in Figure 4, where *equality modulo axioms* $\cong_{\mathsf{K}}$ is specified by a constant symbol Eq_{K}, a notation $x \cong_{\mathsf{K}} y :\leftrightarrow \langle x, y\rangle \in \mathsf{Eq}_{\mathsf{K}}$, and the axiom in Figure 5.

The specification of the quotient sort $\mathsf{K}/\cong_{\mathsf{K}}$ corresponds to the attributes `assoc` and `unit` of $\curvearrowright$.

`syntax K [hook(K.K)]`	$:\leftrightarrow$	$\exists y{:}\mathsf{Sort.K} = y$
`syntax KItem [hook(K.KItem)]`	$:\leftrightarrow$	$\exists y{:}\mathsf{Sort.KItem} = y$
`syntax K ::= KItem`	$:\leftrightarrow$	$\top_{\mathsf{KItem}} \subseteq \top_{\mathsf{K}}$
`syntax K ::= ".K" [symbol(dotK)]`	$:\leftrightarrow$	$\exists z{:}\mathsf{K.dotK} = z$
`syntax K ::= K "~>" K [symbol(↷), assoc, unit(dotK)]`	$:\leftrightarrow$	$x \curvearrowright y :\leftrightarrow (\curvearrowright\ x\ y)$ $\forall x, y{:}\mathsf{K}.\exists z{:}\mathsf{K}.x \curvearrowright y = z$ $\forall x, y{:}\mathsf{K}.x \curvearrowright y \neq \mathsf{dotK}$ $\forall x, x', y, y'{:}\mathsf{K}.x \curvearrowright y = x' \curvearrowright y' \rightarrow x = x' \wedge y = y'$ the axioms of the quotient sort $\mathsf{K}/\cong_{\mathsf{K}}$

Fig. 4. Axioms for the K datatype

$\mathsf{Eq_K} = \mu R{:}\mathsf{K}\otimes\mathsf{K}.\exists x{:}\mathsf{K}.\langle x, x\rangle \vee$	/* reflexive */
$\exists x, y{:}\mathsf{KItem}.\langle x, y\rangle \wedge x \cong_{\mathsf{KItem}} y \vee$	/* $\cong_{\mathsf{KItem}} \subseteq \cong_{\mathsf{K}}$ */
$\exists x, y, z{:}\mathsf{K}.(x \curvearrowright y) \curvearrowright z \cong_{\mathsf{K}} x \curvearrowright (y \curvearrowright z)$	/* associative */
$\exists x{:}\mathsf{K}.x \curvearrowright \mathsf{dotK} \cong_{\mathsf{K}} x$	/* right unit */
$\exists y{:}\mathsf{K.dotK} \curvearrowright y \cong_{\mathsf{K}} y$	/* left unit */
$(\mathsf{intension}\ R)^{-1} \vee$	/* symmetric */
$(\mathsf{intension}\ R) \circ (\mathsf{intension}\ R) \vee$	/* transitive */
$\exists x_1, x_2, y_1, y_2{:}\mathsf{K}.\langle x_1 \curvearrowright y_1, x_2 \curvearrowright y_2\rangle \wedge \langle x_1, x_2\rangle \in R \wedge \langle y_1, y_2\rangle \in R$	/* congruence */

Fig. 5. Definition of $\mathsf{Eq_K}$

4.4 Configurations

The formal syntax for configuration declarations is quite complex, as it allows one to specify various structures needed to define the semantics of programming languages or software systems.

The main component of a configuration is that of a cell, which can be singleton, optional, multiple, and may include other cells. Here we include only the $\mathbb{ML}$ denotation of single cells that include other cells. The specification is given in Figure 6, where S_i is the sort of the cell C_i, i.e., $S_i = \mathsf{Cell}\langle name(C_i)\rangle$). The specification includes a return sort $\mathsf{Cell}\langle cn\rangle$ and a constructor $\langle\!|cn|\!\rangle$ for the content of the cell.

The denotations of cells having the attributes `<cn multiplicity = *, type = ...` are more complex because they have to include specifications of specific data structures mentioned by the `type` attribute, which can be Set, List, or Map. For example, the following cell declaration from Figure 3

```
<threads>
  <thread multiplicity = "*" type = "List">  ... </thread>
</threads>
```

<cn> $C_1 \dots C_k$ </cn>
an instance of
<cell> ::=
  <single-cell>

$$
\begin{array}{rl}
 & \exists y{:}\mathsf{Sort}.\mathsf{Cell}\langle cn\rangle = y \\
 & \exists z.\langle\!|cn|\!\rangle = z \\
 & \forall x_1{:}S_1, \dots, x_k{:}S_k.\exists z{:}\mathsf{Cell}\langle cn\rangle.\langle\!|cn|\!\rangle(x_1, \dots, x_k) = z \\
:\leftrightarrow\ & \forall x_1{:}S_1, \dots, x_k{:}S_k.\forall y_1{:}S_1, \dots, y_k{:}S_k. \\
 & \qquad \langle\!|cn|\!\rangle(x_1, \dots, x_k) = \langle\!|cn|\!\rangle(y_1, \dots, y_k) \rightarrow \\
 & \qquad x_1 = y_1 \wedge \dots \wedge x_k = y_k \\
 & \top_{\mathsf{Cell}\langle cn\rangle} = \exists x_1{:}S_1, \dots, x_k{:}S_k.\langle\!|cn|\!\rangle(x_1, \dots, x_k)
\end{array}
$$

Fig. 6. Axioms for single cells

requires that the contents of the cell <threads> be of sort $\mathsf{ListCell}\langle\mathsf{thread}\rangle$.

Example 3. Here are the axioms for the <T> cell from the configuration of the Minsky Machine (Figure 2):

<T> <k> ... </k> <counters> ... </counters> <stmts> ... </stmts>

$$
:\leftrightarrow
$$

$$
\mathsf{Cell}\langle\mathsf{T}\rangle \in \top_{\mathsf{Sort}} \qquad \exists z.\langle\!|\mathsf{T}|\!\rangle = z
$$

$$
\forall x_1{:}\mathsf{Cell}\langle\mathsf{k}\rangle.\forall x_2{:}\mathsf{Cell}\langle\mathsf{counters}\rangle.\forall x_3{:}\mathsf{Cell}\langle\mathsf{stmts}\rangle.\exists z{:}\mathsf{Cell}\langle\mathsf{T}\rangle.\langle\!|\mathsf{T}|\!\rangle(x_1, x_2, x_3) = z
$$

$$
\forall x_1{:}\mathsf{Cell}\langle\mathsf{k}\rangle.\forall x_2{:}\mathsf{Cell}\langle\mathsf{counters}\rangle.\forall x_3{:}\mathsf{Cell}\langle\mathsf{stmts}\rangle.\langle\!|\mathsf{T}|\!\rangle(x_1, x_2, x_3){=}\langle\!|\mathsf{T}|\!\rangle(y_1, y_2, y_3) \rightarrow x_1{=}y_1 \wedge x_2{=}y_2 \wedge x_3{=}y_3
$$

$$
\top_{\mathsf{Cell}\langle\mathsf{T}\rangle} = \langle\!|\mathsf{T}|\!\rangle(\top_{\mathsf{Cell}\langle\mathsf{k}\rangle}, \top_{\mathsf{Cell}\langle\mathsf{counters}\rangle}, \top_{\mathsf{Cell}\langle\mathsf{stmts}\rangle})
$$

The specification of the <k> cell, e.g., is similar but its content is of sort K:

<k> ... </k>

$$
:\leftrightarrow
$$

$$
\mathsf{Cell}\langle\mathsf{k}\rangle \in \top_{\mathsf{Sort}} \qquad \exists z.\langle\!|\mathsf{k}|\!\rangle = z \qquad \forall x{:}\mathsf{K}.\exists z{:}\mathsf{Cell}\langle\mathsf{k}\rangle.\langle\!|\mathsf{k}|\!\rangle(x) = z
$$

$$
\top_{\mathsf{Cell}\langle\mathsf{k}\rangle} = \exists x{:}\mathsf{K}.\langle\!|\mathsf{k}|\!\rangle(x) \qquad \forall x, y{:}\mathsf{K}.\langle\!|\mathsf{k}|\!\rangle(x) = \langle\!|\mathsf{k}|\!\rangle(y) \rightarrow x = y
$$

4.5 Semantic Rewrite Rules

These rules define the dynamic semantics of a language/system: each such a rule specifies a one-step execution. We first explain the most general form of a rule. Assuming that we already have

cfg => *cfg′* an instance of <rule-body> $:\leftrightarrow \forall \bar{x}.\mathit{cfg}^{\mathbb{ML}} \rightarrow \bullet\mathit{cfg}'^{\mathbb{ML}}$

then

rule *cfg* => *cfg′* requires *precond* ensures *postcond* an instance of

```
<rule-declaration> ::=
    "rule" <rule-body> "requires" <condition>
                       "ensures" <condition>
```

$$
:\leftrightarrow
$$

$$
\forall \bar{x}.(\mathit{cfg}^{\mathbb{ML}} \wedge \mathit{precond}^{\mathbb{ML}}) \rightarrow \bullet(\mathit{cfg}'^{\mathbb{ML}} \wedge \mathit{postcond}^{\mathbb{ML}})
$$

If either the precondition or the postcondition (or both) is omitted from the rule declaration, it can be assumed to be equal to $\top$.

We explain in the following how $\forall \bar{x}.cfg^{\mathbb{ML}} \to \bullet cfg'^{\mathbb{ML}}$ is obtained for various kinds of rewrites cfg `=>` cfg'. Here is a place where the axiomatic definition of contexts plays a major role.

K-Contextual Rewrite These rules describe changes in `<k>`-cells without mentioning any cell/context.

$t \Rightarrow t'$ an instance of `<rule-body> ::= <k-term> "=>" <k-term>`

$$:\leftrightarrow$$

$$\forall C : \mathsf{Context}^{\mathsf{Cell}\langle \mathsf{T}\rangle}_{\mathsf{Cell}\langle \mathsf{k}\rangle}.\forall \kappa{:}\mathsf{K}.C[\langle\!|\mathsf{k}|\!\rangle(t^{\mathbb{ML}} \curvearrowright \kappa)] \to \bullet C[\langle\!|\mathsf{k}|\!\rangle(t'^{\mathbb{ML}} \curvearrowright \kappa)]$$

Example 4. For the $\mathbb{K}$ definition in Figure 2 we have

`rule decjz(0, L1, _) => L1`

$$:\leftrightarrow$$

$$\forall C{:}\mathsf{Context}^{\mathsf{Cell}\langle \mathsf{T}\rangle}_{\mathsf{Cell}\langle \mathsf{k}\rangle}.\forall l_1, l_2{:}\mathsf{Label}.(C \circ C_\kappa)[\mathsf{decjz}(0, l_1, l_2)] \to \bullet(C \circ C_\kappa)[l_1]$$

Local Rewrites These rules specify one-step executions by mentioning only those configuration components that are changed (using abstraction mechanism).

We start by specifying the components of the the $\mathbb{ML}$ denotation given by the local rewrites in a single cell:

$\langle cn\rangle\, c[t_1 \Rightarrow t'_1]\cdots[t_k \Rightarrow t'_k]\,\langle/cn\rangle$ an instance of

`<local-changes-under-local-multi-context>`

$$:\leftrightarrow$$

$$\begin{array}{l}\gamma\square_1{:}s_1 \ldots \gamma\square_k{:}s_k.c^{\mathbb{ML}}{:}\mathsf{Context}^{\mathsf{Cell}\langle cn\rangle}_{s_1,\ldots,s_k}\\ t_1^{\mathbb{ML}}, \ldots, t_k^{\mathbb{ML}}\\ {t'_1}^{\mathbb{ML}}, \ldots, {t'_k}^{\mathbb{ML}}\end{array}$$

where s_i is the sort of t_i and t'_i and local rewrites $t_i \Rightarrow t'_i$ in c are replaced by the corresponding holes $\square_i$ in $c^{\mathbb{ML}}$.

Now we extend the specification to all local rewrites declared in a rule body:

$\langle cn^1\rangle c^1[t^1_1 \Rightarrow {t'}^1_1]\ldots[t^1_{k_1} \Rightarrow {t'}^1_{k_1}]\langle/cn^1\rangle \ldots \langle cn^\ell\rangle c^\ell[t^\ell_1 \Rightarrow {t'}^\ell_1]\ldots[t^\ell_{k_\ell} \Rightarrow {t'}^\ell_{k_\ell}]\langle/cn^\ell\rangle$

an instance of

`<rule-body> ::= <local-changes-under-local-multi-contexts>`

$$:\leftrightarrow$$

$$\begin{array}{l}\forall C{:}\mathsf{Context}^{\mathsf{Cell}\langle \mathsf{T}\rangle}_{\mathsf{Cell}\langle cn_1\rangle,\ldots,\mathsf{Cell}\langle cn_\ell\rangle}.\\ \quad (C \odot \langle c_1^{\mathbb{ML}}, \ldots, c_\ell^{\mathbb{ML}}\rangle)[t_1^{1\,\mathbb{ML}}, \ldots, t_{k_1}^{1\,\mathbb{ML}}, \ldots, t_1^{\ell\,\mathbb{ML}}, \ldots, t_{k_\ell}^{\ell\,\mathbb{ML}}] \to\\ \quad \bullet(C \odot \langle c_1^{\mathbb{ML}}, \ldots, c_\ell^{\mathbb{ML}}\rangle)[{t'}_1^{1\,\mathbb{ML}}, \ldots, {t'}_{k_1}^{1\,\mathbb{ML}}, \ldots, {t'}_1^{\ell\,\mathbb{ML}}, \ldots, {t'}_{k_\ell}^{\ell\,\mathbb{ML}}]\end{array}$$

where $c_i^{\mathbb{ML}}$ are defined as above.

Example 5. For the $\mathbb{K}$ definition in Figure 2 we have

```
rule <k> inc(C:Counter, L:Label) => L ... </k>
     <counters> ... C | -> ( V => V +Int 1) </counters>
```

$$:\leftrightarrow$$

$$\forall C{:}\mathsf{Context}^{\mathsf{Cell}\langle\mathsf{T}\rangle}_{\mathsf{Cell}\langle\mathsf{k}\rangle,\mathsf{Cell}\langle\mathsf{counters}\rangle}.\forall c{:}\mathsf{Counter}.\forall l{:}\mathsf{Label}.\forall m{:}\mathsf{Map}.\forall v{:}\mathsf{Int}.$$
$$(C \odot \langle C_1, C_2\rangle)[\mathsf{inc}(c,l), v] \to \bullet(C \odot \langle C_1, C_2\rangle)[l, v+1]$$

where $C_1 := \gamma\Box{:}\mathsf{K}.\langle\!|\mathsf{k}|\!\rangle(\Box \curvearrowright \kappa)$, $C_2 := \gamma\Box{:}\mathsf{Int}.\langle\!|\mathsf{counters}|\!\rangle(m\ \mathsf{MAP}\ c \mapsto \Box)$. Here, MAP is the concatenation operation on the sort Map and $c \mapsto \Box$ represents the singleton map with key c and value $\Box$, which are part of the theory of builtins.

5 Related Work

The 𝕂 Framework's rule-based approach is conceptually similar to Structural Operational Semantics (SOS) [28]. However, 𝕂 introduces powerful abstraction mechanisms, such as implicit configuration contexts, inspired from evaluation contexts [14,13]. The practical importance of providing a solid formal foundation for 𝕂 is emphasized by its widespread use in developing complete formal definitions for numerous real-world languages, including C [15], Java [2], JavaScript [25], x86 [12], LLVM [20], as well as emerging blockchain languages such as EVM [16] and IELE [17]. Recently, 𝕂 was used to generate proof certificates for a language-agnostic interpreter, where each (concrete) execution of a program is certified by a machine-checkable mathematical proof [4,21].

The first version of the matching logic was used to specify static properties of programs in reachability logic [30,11]. A sound and complete Hilbert-style proof system of (structural) matching logic is given in [29]. The matching logic was then extended with a least fixpoint-binder μ [9], which subsumes not only reachability logic, but also a variety of common logics/calculi that are used to reason about fixpoints and induction. These versions follow a many-sorted approach and are used in the design of the intermediate representation Kore. In this paper, we use the functional variant of matching logic [7,6] that adopts a minimalist design and defines only the simplest building blocks, from which more complex and notationally heavier concepts can be axiomatized as theories, and intended to be used as a basic reference.

A central theme of our work is the formal treatment of contexts. While the notion of a context as a term-with-a-hole is fundamental to term rewriting, it is typically treated as a metatheoretical concept. Our work reifies contexts as first-class objects within the logic, an approach that shares conceptual similarities with other formalisms such as higher-order abstract syntax (HOAS) in frameworks such as Twelf [26], refinement calculus [1,24], and program transformation languages such as Stratego [37]. Our formalization in the paper opens the door for providing a common matching logic foundation for these related approaches.

While in our approach we chose to represent binders using the intension technique [8], another possibility would have been to use nominal logic, which was axiomatized as an 𝕄𝕃 theory [10,35]. Regardless of this choice, the use of context for denoting 𝕂 definitions would work in the same way, after establishing the main properties of the plugging operation.

There have been other theoretical attempts to formalize $\mathbb{K}$, e.g., using double-pushout graph transformations [36], Isabelle [19], Rocq [23]; however, these approaches introduce a significant representational distance from the original definitions. In contrast, our approach provides a direct denotational semantics, mapping a $\mathbb{K}$ definition to a formal theory in $\mathbb{ML}$. This is distinct from the classic denotational semantics [34], which maps programs to mathematical domains.

The ultimate motivation for this work is to enable the formal verification of the kompile tool, placing it in the broader context of verified software toolchains. The most notable project in this area is CompCert [18], a formally verified C compiler that guarantees semantic preservation from source code to assembly. While CompCert focuses on verifying the translation of programs, our work provides the basis for addressing the unique challenge of verifying the translation of language definitions.

6 Conclusion

In this paper, we have presented a formal denotational semantics for the $\mathbb{K}$ frontend, establishing a direct and transparent link between high-level $\mathbb{K}$ language definitions and their underlying foundation in matching logic ($\mathbb{ML}$). We have systematically demonstrated how each component of a $\mathbb{K}$ definition can be formally translated into a corresponding $\mathbb{ML}$ theory.

Our approach is centered on a novel, reified theory of contexts within $\mathbb{ML}$, which proved essential for elegantly capturing $\mathbb{K}$'s complex context-dependent features, including the configuration abstraction mechanism and the heating/-cooling rules for dynamic strictness attributes. By formalizing these concepts, we replace the opaque "magic" of kompile with a clear and verifiable specification.

This work provides the formal foundation necessary to verify the correctness of the $\mathbb{K}$ toolchain. An immediate direction for future work is to leverage this specification to formally prove that the Kore generated by the kompile tool is a correct refinement of the $\mathbb{ML}$ theory we have defined. To this end, we would first need a definition of the Kore language as an $\mathbb{ML}$ theory, which is well-known and was for example formalized in the Metamath proof system [4]. Then, we would need to prove that the Kore theory generated by kompile is indeed a refinement of the $\mathbb{ML}$ theory defined using our approach.

Acknowledgments. The authors would like to thank the reviewers for their insightful and constructive comments. Many thanks to Iulia Bastys for helping in the improvement of the presentation of this paper. This study was funded by Pi Squared.

References

1. Back, R.J., von Wright, J.: Refinement Calculus: A Systematic Introduction. Graduate Texts in Computer Science, Springer-Verlag (1998). https://doi.org/10.1007/978-1-4612-1672-7

2. Bogdănaş, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15). pp. 445–456. ACM (2015)
3. Chen, X., Cheval, H., Lucanu, D., Rou, G.: A Matching Logic theory of multihole contexts (2026), submitted
4. Chen, X., Lin, Z., Trinh, M.T., Roşu, G.: Towards a trustworthy semantics-based language framework via proof generation. In: Proceedings of the 33rd International Conference on Computer-Aided Verification. ACM (July 2021)
5. Chen, X., Lucanu, D., Roşu, G.: Initial algebra semantics in matching logic. Tech. rep., University of Illinois at Urbana-Champaign (Jul 2020), http://hdl.handle.net/2142/107781
6. Chen, X., Lucanu, D., Roşu, G.: Matching logic explained. J. Log. Algebraic Methods Program. **120**, 100638 (2021). https://doi.org/10.1016/J.JLAMP.2021.100638
7. Chen, X., Roşu, G.: Applicative matching logic: Semantics of K. Tech. rep., University of Illinois at Urbana-Champaign (July 2019), http://hdl.handle.net/2142/104616
8. Chen, X., Roşu, G.: A general approach to define binders using matching logic. In: Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'20). vol. 4, pp. 1–32. ACM/IEEE (Aug 2020), https://doi.org/10.1145/3408970
9. Chen, X., Rou, G.: Matching μ-logic. In: 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13 (2019). https://doi.org/10.1109/LICS.2019.8785675
10. Cheney, J., Fernández, M.: Nominal matching logic. In: PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022. pp. 5:1–5:15. ACM (2022). https://doi.org/10.1145/3551357.3551375
11. Ştefănescu, A., Ciobâcă, Ş., Mereuţă, R., Moore, B.M., Şerbănuţă, T.F., Roşu, G.: All-path reachability logic. In: Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14). vol. 8560, pp. 425–440. Springer (2014)
12. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19). ACM (2019)
13. Felleisen, M., Findler, R.B., Flatt, M.: Semantics engineering with PLT Redex. Mit Press (2009)
14. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science **103**(2), 235–271 (1992). https://doi.org/https://doi.org/10.1016/0304-3975(92)90014-7
15. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of C. In: Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 336–345. ACM (2015)
16. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: KEVM: A complete semantics of the Ethereum virtual machine. In: Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18). IEEE (2018), http://jellopaper.org

17. Kasampalis, T., Guth, D., Moore, B., Şerbănuţă, T.F., Zhang, Y., Filaretti, D., Şerbănuţă, V., Johnson, R., Roşu, G.: IELE: A rigorously designed language and tool ecosystem for the blockchain. In: Proceeding of the 23rd International Symposium on Formal Methods (FM'19) (2019)
18. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
19. Li, L., Gunter, E.: IsaK-static A complete static semantics of K. In: Formal Aspects of Component Software. pp. 196–215. Springer (2018)
20. Li, L., Gunter, E.L.: K-LLVM: A Relatively Complete Semantics of LLVM IR. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 7:1–7:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.7
21. Lin, Z., Chen, X., Trinh, M.T., Wang, J., Roşu, G.: Generating proof certificates for a language-agnostic deductive program verifier. In: Proceedings of OOPSLA 2023. ACM (July 2023)
22. Minsky, M.L.: Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines. Annals of Mathematics **74**(3), 437–455 (1961), http://www.jstor.org/stable/1970290
23. Moore, B., Peña, L., Roşu, G.: Program verification by coinduction. In: Proceedings of the 27th European Symposium on Programming (ESOP'18). pp. 589–618. Springer (2018)
24. Morgan, C.: Programming from Specifications. International Series in Computer Science, Prentice Hall International, 2nd edn. (1998)
25. Park, D., Ştefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 346–356. ACM (2015)
26. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Proceedings of the 16th International Conference on Automated Deduction (CADE-16). Lecture Notes in Computer Science, vol. 1632, pp. 202–206. Springer (1999)
27. Pi Squared: Proof of Proof: Verifiable computing for all languages. https://pi2.network/papers/proof-of-proof-whitepaper (2025)
28. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. FN-19, DAIMI, Aarhus University (1981)
29. Roşu, G.: Matching logic. Logical Methods in Computer Science **13**(4), 1–61 (December 2017). https://doi.org/https://doi.org/10.23638/LMCS-13(4:28)2017
30. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13). pp. 358–367. IEEE (2013)
31. Runtimeverification: K Language Features. https://github.com/runtimeverification/k/blob/master/k-distribution/include/kframework/builtin/kast.md
32. Runtimeverification: K User Manual. https://github.com/runtimeverification/k/blob/master/docs/user_manual.md
33. Runtimeverification: Basic builtin types in K. https://github.com/runtimeverification/k/blob/master/k-distribution/include/kframework/builtin/domains.md (2024)

34. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages. In: Proceedings of the Symposium on Computers and Automata. pp. 19–46. Polytechnic Institute of Brooklyn Press, New York, NY, USA (1971)
35. Sebe, M., Fernández, M., Cheney, J.: Nominal matching logic with fixpoints. In: Stark, K., Timany, A., Blazy, S., Tabareau, N. (eds.) Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2025, Denver, CO, USA, January 20-21, 2025. pp. 17–33. ACM (2025). https://doi.org/10.1145/3703595.3705872, https://doi.org/10.1145/3703595.3705872
36. Şerbănuţă, T.F., Roşu, G.: A truly concurrent semantics for the K framework based on graph transformations. In: Proceedings of the 6th International Conference on Graph Transformation (ICGT'12). pp. 294–310. Springer (2012)
37. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. In: International Conference on Rewriting Techniques and Applications (RTA'01). Lecture Notes in Computer Science, vol. 2051, pp. 357–361. Springer (2001)