A semantics of K into Dedukti

Amélie Ledein, Valentin Blot, Catherine Dubois

A semantics of $\mathbb{K}$ into Dedukti

Amélie Ledein
Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Valentin Blot
Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Catherine Dubois
ENSIIE, IP Paris, Samovar, France

Abstract

$\mathbb{K}$ is a semantical framework for formally describing the semantics of programming languages thanks to a BNF grammar and rewriting rules on configurations. It is also an environment that offers various tools to help programming with the languages specified in the formalism. For example, it is possible to execute programs thanks to the generated interpreter, or to check some properties of them thanks to the provided automatic theorem prover called the KProver. $\mathbb{K}$ is based on Matching Logic, a first-order logic with an application between formulas and a fixed-point, extended with symbols to encode equality, typing and rewriting. This specific Matching Logic theory is called Kore.

Dedukti is a logical framework having for main goal the interoperability of proofs between different formal proof tools. Several translators to Dedukti exist or are under development, in order to automatically translate formalizations written, for instance, in Coq or PVS. Dedukti is based on the $\lambda\Pi$-calculus modulo theory, a λ-calculus with dependent types and extended with a primitive notion of computation defined by rewriting rules. The flexibility of this logical framework allows to encode many theories ranging from first-order logic to the Calculus of Constructions.

In this article, we present a paper formalization of the translation from $\mathbb{K}$ semantics into Kore, and a paper formalization and an automatic translation tool, called KaMeLo, from Kore to Dedukti in order to execute programs in Dedukti.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases Programming language, Semantics, Rewriting, Logical framework, Type theory.

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.23

Funding *Amélie Ledein*: Digicosme and EuroProofNet.

Acknowledgements We want to thank the $\mathbb{K}$ team, especially Andrei Arusoaie, Xiaohong Chen, Denisa Diaconescu, Everett Hildenbrandt, Zhengyao Lin, Dorel Lucanu, Ana Pantilie and Traian-Florin Serbanuta for their prompt responses to our many questions.

1 Introduction

The main objective of formal methods is to obtain greater confidence in programs. But before verifying a program, it must be written in a programming language whose syntax and semantics we must know precisely. Therefore, we must first have a formalization of the semantics of the programming language used to write the program we wish to verify. Several tools make it possible to write formal semantics such as Centaur [8], ASF+SDF [23], Ott [21], Sail [5], Lem [18] or $\mathbb{K}$ [20, 4]. In this article, we are only interested in the latter, since there are currently a large number of programming language semantics written in $\mathbb{K}$ such as Java [7], C [14] or JavaScript [19].

$\mathbb{K}$ is a semantical framework which offers many facilities for writing a semantics, such as attributes to specify evaluation strategies. Once the semantics of a language $\mathcal{L}$ has been specified, $\mathbb{K}$ allows to execute a program $\mathcal{P}$ written in $\mathcal{L}$ but also the possibility of verifying

some properties - expressed in the form of reachability properties - on the program $\mathcal{P}$ using the automatic theorem prover KProver [22]. As it is possible to automatically translate $\mathbb{K}$ semantics into a Matching Logic theory named Kore, the particularity of the $\mathbb{K}$ framework is to see any semantics $\mathcal{S}$ of a programming language $\mathcal{L}$ as a logical theory $\Gamma^{\mathcal{L}}$. That's the reason we look at the translation of $\mathbb{K}$ into Dedukti, which is a logical framework based on λΠ-calculus modulo theory having for main goal the interoperability of proofs between different formal proof tools.

In this article, we are more particularly interested in the translation towards Dedukti of any semantics $\mathcal{S}$ of a programming language $\mathcal{L}$ written in $\mathbb{K}$ in order to execute in Dedukti programs written in $\mathcal{L}$. Our first contribution is a paper formalization of the translation from $\mathbb{K}$ semantics into Kore. As no article has been yet published on this translation, this contribution was elaborated by reverse engineering on Kore files as well as thanks to discussions with the $\mathbb{K}$ team. Independently, we formalize transformations similar to what was done in IsaK [16, 17], which is a formalization in Isabelle – but not based on Kore – of the static and dynamic semantics of $\mathbb{K}$. In addition, the second contribution is a paper formalization and an automatic tool, called KaMeLo, from Kore to Dedukti in order to execute programs in Dedukti. The general overview of the translation pipeline presented in this article is available on Figure 1.

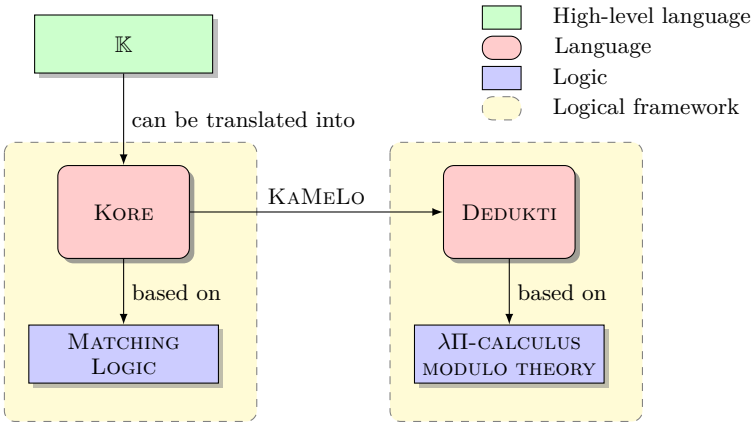

Figure 1 Overview of the translation pipeline presented in this article

In the long-term, the translation from $\mathbb{K}$ to Dedukti would make it possible to execute a program $\mathcal{P}$ written with the formalized language $\mathcal{L}$ within Dedukti, verify the proofs established by the KProver, or even make this proof with Dedukti if the KProver has failed, and also formally check meta-properties on the language $\mathcal{L}$. This long-term goal can be seen as a new pipeline for program verification, which is parametrized by a programming language and leaves the user free to choose the proof assistant they wish.

This article is structured as follows: first, we explain how to write a $\mathbb{K}$ semantics (Section 2). Then, we present a mathematical structure $\mathcal{M}$ to abstract $\mathbb{K}$, in order to formalize some internal transformations that $\mathbb{K}$ does on an semantics (Section 3). Thanks to the mathematical structure $\mathcal{M}$, we formalize a computational translation $\mathcal{T}$ into λΠ-calculus modulo theory (Section 4). Finally, we present our implementation of $\mathcal{T}$ (Section 5).

In the following, the keywords of a language or what is native in a language will be distinguished by color. The language of Dedukti will be distinguished by `a blue color`, the language of $\mathbb{K}$ by `an orange color` and the language of Kore by `a red color`. These facilitate reading, but are not necessary for understanding.

2 What is the $\mathbb{K}$ framework?

This section introduces the $\mathbb{K}$ framework by explaining how to write a semantics using a small example, and then by presenting the diversity of $\mathbb{K}$ features. This section ends with the $\mathbb{K}$ grammar that we consider in the rest of this article.

2.1 A first $\mathbb{K}$ semantics

In this subsection, we show an example, based on booleans and two lazy symbols, to illustrate how to write a semantics in $\mathbb{K}$. As usual, the first step to formalize a semantics is to define the syntax and then the semantics associated to the syntax.

2.1.1 Define the syntax of a language

Defining the syntax of a language in $\mathbb{K}$ is similar to writing a *BNF grammar*. This is done in the module LAZY-SYNTAX (Figure 2 - lines 1 to 9) with the definition of booleans and two lazy symbols. A terminal symbol will be written between quotes, as for example "||", and anything else in **bold** will therefore be a non-terminal symbol. In order to make the syntax parseable, it is possible to use *attributes*, i.e. keywords between square brackets, allowing to specify the associativity (`left`, `right`, `non-assoc`) or to add parentheses to the language (`bracket`). We explain as we go along the other attributes.

```
1  module LAZY-SYNTAX
2        syntax MyBool ::= "true"                [ constructor ]
3                        | "false"               [ constructor ]
4        syntax KResult ::= MyBool
5        syntax BExp ::= MyBool
6          | BExp "||" BExp                      [ left, function ]
7          | BExp "&&" BExp                      [ left, constructor, strict(1) ]
8          | "(" BExp ")"                        [ bracket ]
9  endmodule
10
11 module LAZY
12       imports LAZY-SYNTAX
13       configuration <k> $PGM : BExp </k>

14       rule false || B => B
15       rule true  || _ => true
16
17       rule true  && B => B
18       rule false && _ => false
19 endmodule
```

Figure 2 Syntax and semantics of booleans and two lazy symbols

Moreover, $\mathbb{K}$ supports extended BNF (EBNF) thanks to `List`, which corresponds to the EBNF operator "*", and `NeList`, which corresponds to the EBNF operator "+".

2.1.2 Define the semantics associated to the syntax

The main ingredients for defining the semantics associated with each element of the syntax are *configurations* and *rewriting rules*. Some attributes are also useful to define other aspects of the semantics such as evaluation strategies. The semantics of a computational lazy disjunction, noted `||`, and of a semantical lazy conjunction, noted `&&`, is defined in the module LAZY (Figure 2 - lines 11 to 19), which imports the syntax module (`imports`).

2.1.2.1 Configurations

A *configuration* models the state of the program and is composed of several *cells*. For example, the configuration $\langle\langle\ x = 10;\ \rangle_k\ \langle\ x \mapsto 0\ \rangle_{env}\rangle$ is composed of two cells, one labelled by k containing the program to be executed, the other labelled by *env* containing the current values of the variables. In the example on Figure 2, the configuration contains only the cell k (line 13). The configuration variable `$PGM` will contain the parsed program given by the user.

2.1.2.2 Rewriting rules

A $\mathbb{K}$ rewriting rule is of the 1st order, and can be conditional, noted `rule` *LHS* `=>` *RHS* `requires` *Cond*, or unconditional, noted `rule` *LHS* `=>` *RHS*. Moreover, a $\mathbb{K}$ rewriting rule can be non-linear, and the variables in the left-hand side (*LHS*) can be omitted using a wildcard (`_`) when they are not used in the right-hand side (*RHS*), as in the rule on line 15 or 18 (Figure 2). Finally, $\mathbb{K}$ supports partial rewriting modulo ACUI, i.e. associativity (`assoc`), commutativity (`comm`), identity (`unit`) and idempotence (`idem`).

As any symbol is either a constructor symbol (`constructor`) or a function symbol (`function`), any $\mathbb{K}$ rewriting rule applies to a configuration or to a part of a configuration. Intuitively, a `function` symbol and its associated rules correspond to computational content, whereas a `constructor` symbol and its associated rules correspond to language semantics. This distinction affects the completion of rewriting rules by $\mathbb{K}$ as illustrated more precisely in the next paragraph.

2.1.2.3 Evaluation strategies

To define an evaluation strategy, i.e. specifying the order in which the sub-expressions are evaluated, it is possible to use *contexts* (`context`) as is conventionally done, but also *context templates* (`context alias`), which allow contexts to be generated automatically rather than systematically writing similar contexts.

There are also two attributes for defining an evaluation strategy: `strict` defines non-deterministic strategies and `seqstrict` defines deterministic strategies from left to right by default. It is also possible to restrict the list of sub-expressions that must be evaluated by giving a list of numbers as done in Figure 2. Indeed, the attribute `strict(1)` forces the evaluation of the first argument of the symbol `&&`, and then it is possible to apply one of the rules on line 17 or 18. To use these attributes, the user needs to define the sort `KResult`, which allows to distinguish final values from expressions thanks to typing. For instance, the line 4 (Figure 2) specifies that a final value is either `false` or `true`.

Whichever way an evaluation strategy is defined, it is translated using *K computations* and *freezers*. A K computation is a potentially nested list of computations to be performed sequentially and built with the constructors `.` and $\curvearrowright$, whereas a freezer is a symbol which encapsulates the part of the computation that shouldn't yet be modified, i.e. the tail of the K computation, while waiting for the head of the K computation to be evaluated. This mechanism is inspired by evaluation contexts [25] and by continuations v $\curvearrowright$ C.
The rewriting rules generated by `strict(1)` (Rules n°1 and n°2, with the attributes `heat` and `cool`) as well as an example of an execution are detailed in Figure 3. Freezers are noted ($\blacklozenge^{nb}_{sym}$ arg) where *sym* is a symbol, *nb* the number of the argument whose value we expect, and *arg* the list of other arguments. As the symbol `&&` has the attribute `constructor`, the rules on lines 17 and 18 (Figure 2) are respectively translated by $\mathbb{K}$ into the rules n°3 and n°4 (Figure 3). In contrast, as the symbol `||` has the attribute `function`, $\mathbb{K}$ does not transform the rules on lines 14 and 15 (Figure 2).

1. rule $\langle\ E_1$ && $E_2 \curvearrowright S\ \rangle_k$ => $\langle\ E_1 \curvearrowright (\circledast^1_{\&\&}\ E_2) \curvearrowright S\ \rangle_k$ requires $\neg$ (isKResult E_1) [heat]
2. rule $\langle\ E_1 \curvearrowright (\circledast^1_{\&\&}\ E_2) \curvearrowright S\ \rangle_k$ => $\langle\ E_1$ && $E_2 \curvearrowright S\ \rangle_k$ requires isKResult E_1 [cool]
3. rule $\langle$ true && $B \curvearrowright S\ \rangle_k$ => $\langle\ B \curvearrowright S\ \rangle_k$
4. rule $\langle$ false && _ $\curvearrowright S\ \rangle_k$ => $\langle$ false $\curvearrowright S\ \rangle_k$

$\langle$ (true && false) && (true && true) $\curvearrowright$. $\rangle_k$
$\hookrightarrow_1$ $\langle$ (true && false) $\curvearrowright$ ($\circledast^1_{\&\&}$ (true && true)) $\curvearrowright$. $\rangle_k$
$\hookrightarrow_3$ $\langle$ false $\curvearrowright$ ($\circledast^1_{\&\&}$ (true && true)) $\curvearrowright$. $\rangle_k$
$\hookrightarrow_2$ $\langle$ false && (true && true) $\curvearrowright$. $\rangle_k$
$\hookrightarrow_4$ $\langle$ false $\curvearrowright$. $\rangle_k$

Figure 3 Translation of the attributes `strict(1)` and an example execution

The translation of the attribute `strict(1)` into rewriting rules is similar in the case of attributes `strict` and `seqstrict`.

In this article, a rewriting rule that has a constructor symbol as its head is called *semantical*, and a rewriting rule that has a function symbol as its head is called *computational*. The attributes `assoc`, `comm`, `unit` and `idem` generate equations, named *equational rules*. A rewriting rule with the attribute `heat` or `cool` is called an *evaluation strategy rule*.

2.2 Additional features

The previous subsection illustrated the main $\mathbb{K}$ features. However, there are many other features, which come from attributes or the $\mathbb{K}$ standard library, in order to bring more precision to a semantics.

2.2.1 Definable features thanks to the attributes

$\mathbb{K}$ has about 70 attributes. $\mathbb{K}$ papers mentioned very few of them and the documentation is not exhaustive and complete. However, many features require the use of attributes. We present here the list of attributes on Figure 4 that we hope to be as exhaustive as possible.

About importation.	$\mathcal{A}_{library}$	$\triangleq$	{ `hook` }
	$\mathcal{A}_{visibility}$	$\triangleq$	{ `public`, `private` }
	$\mathcal{A}_{backend}$	$\triangleq$	{ `symbolic`, `concrete`, `kast`, `kore` }
About parsing.	$\mathcal{A}_{parsing}$	$\triangleq$	{ `left`, `right`, `non-assoc`, `prefer`, `avoid`, `applyPriority` }
	$\mathcal{A}_{sort}$	$\triangleq$	{ `token`, `locations`, `hook` }
	$\mathcal{A}_{token}$	$\triangleq$	{ `prefer`, `prec(`*nb*`)`, `hook` }
About printing.	$\mathcal{A}_{printing}$	$\triangleq$	{ `color`, `colors`, `symbol`, `klabel`, `bracketLabel`, `format`, `latex`, `unused` }
About symbol.	$\mathcal{A}_{family}$	$\triangleq$	{ `constructor`, `function`, `token`, `bracket`, `macro` }
	$\mathcal{A}_{property}$	$\triangleq$	{ `injective`, `total`, `freshGenerator`, `binder` }
	$\mathcal{A}_{strategy}$	$\triangleq$	{ `strict`, `seqstrict`, `result`, `hybrid` }
About cell.	$\mathcal{A}_{structure}$	$\triangleq$	{ `multiplicity=` {`"+"` | `"*"` }, `type="` ⟨**sort**⟩`"` }
	$\mathcal{A}_{console}$	$\triangleq$	{ `exit="` ⟨**sort**⟩`"`, `stream=` {`"stdin"` | `"stdout"` | `"stderr"` } }
About rewriting.	$\mathcal{A}_{modulo}$	$\triangleq$	{ `assoc`, `comm`, `unit`, `idem` }
	$\mathcal{A}_{rule}$	$\triangleq$	{ `heat`, `cool`, `priority(`*nb*`)`, `owise`, `anywhere`, `unboundVariables` }
About proof.	$\mathcal{A}_{\text{KPROVER}}$	$\triangleq$	{ `symbolic`, `concrete`, `all-path`, `one-path`, `simplification`, `trusted`, `smtlib`, `smt-lemma`, `smt-hook`, `memo` }

Figure 4 The exhaustive list of $\mathbb{K}$ attributes

About importation. What comes from the $\mathbb{K}$ standard library, briefly presented in Section 2.2.2, has the attribute `hook`. Furthermore, we can specify the visibility of a module or an import ($\mathcal{A}_{visibility}$) or that a module is only useful for some backends ($\mathcal{A}_{backend}$).

About parsing. The user can precise constraints about parsing such as the associativity of symbols (`left`, `right`, `non-assoc`) but also to reject cases of parsing ambiguity (`prefer`, `avoid`, `applyPriority`). Moreover, it is possible to type a part of the AST by declaring particular identifiers (`token`) that can be used later in the semantics. The precedence of a token is given by the attribute `prec(`*`nb`*`)`. Sorts with the attribute `token` are only composed of symbols with the attribute `function` or `token`, and only these sorts can be composed of `token` symbols. Finally, $\mathbb{K}$ is able to insert file, line and column meta-data into the parse tree on a subtree of type s when parsing, when the sort s has the attribute `locations`.

About printing. There are also some attributes to change colors of printing in the console (`color`, `colors`), the names (`symbol`, `klabel`, `bracketLabel`) and the printing (`format`) of the symbols and to define a latex name (`latex`). Moreover, $\mathbb{K}$ will warn the user if a symbol is declared but not used in any of the rules. The user can disable this warning by adding the attribute `unused` to the concerned symbol or cell.

About symbol. Compare to one definable with the the attributes `strict` and `seqstrict` (Section 2.1.2.3), it possible to develop more complex strategies thanks to the attributes `result` and `hybrid`, such as being able to consider lists of values as being values.

A symbol can be (1) a *constructor*, (2) a *function*, (3) a *token*, (4) a *bracket* or (5) a *macro* ($\mathcal{A}_{family}$). Functions can be defined as injective (`injective`) or total (`total`, formerly called `functional`). Moreover, it is possible to request $\mathbb{K}$ to generate fresh values and used them with fresh variables `!`Var (`freshGenerator`) and also to define binder (`binder`).

About cell. The user can choose if a cell is optional or not (`multiplicity="?"`), or can appear several times (`multiplicity="*"`). By this way, the user defines a set of cells, which the type can be defined as `List`, `Set` or `Map` thanks to the attribute `type`. Moreover, each cell can have a console exit value (`exit`) or can print on the standard stream (`stream`).

About rewriting. In the theory of rewriting, the rewriting rules are applied in any order but $\mathbb{K}$ allows to associate a priority to each rule (`priority(`*`nb`*`)`) or to indicate that a rule applies only if no other can apply (`owise`). Moreover, the attribute `anywhere` can be used to prevent $\mathbb{K}$ from automatically completing the configuration in a rewriting rule. Finally, it is also possible to allow variables to be unbound by the left-hand side of a rewriting rule thanks to the attribute `unboundVariables` or with unbound variables `?`Var.

The semantics of most of these attributes are specified in the rest of this article.

2.2.2 Definable features thanks to the $\mathbb{K}$ standard library.

The $\mathbb{K}$ standard library is composed of the following files: `prelude.md` is imported into any $\mathbb{K}$ definition and containing only two lines which import the following two files, `domains.md` which defines several usual data structures such as `Bytes`, `Array`, `Map`, `Set`, `List`, `Bool`, `Int`, `String` and `Float`, `kast.md` which corresponds to the syntax of $\mathbb{K}$, `rat.md` is an implementation of the rational integers, `substitution.md` is an implementation allowing substitution (required by the attribute `binder`), `unification.md` is an implementation allowing unification, `ffi.md` which allows C functions to be called and `json.md` which allows JSON files to be read. The three following symbols are also defined: `.` : `K` [`constructor`], $\curvearrowright$: `KItem` $\rightarrow$ `K` $\rightarrow$ `K` [`constructor`] and `inj` : $\forall$ ($From$, To : `K`), $From \rightarrow To$. We abstract the content of the $\mathbb{K}$ standard library by the sets $\mathcal{S}ort_{lib}$, $\mathcal{R}el_{lib}$ $\mathcal{S}ym_{lib}$ and $\mathcal{R}_{lib}$.

2.3 A $\mathbb{K}$ grammar

The $\mathbb{K}$ grammar that we consider in this article is available on Figure 6. A $\mathbb{K}$ file can contain several modules (`module`/`endmodule`). It is possible to import files into another file (`requires` or `require`) or to import one or more modules into another module (`imports` or `import`). Additionally, each attribute is associated to a module ($\mathcal{A}_{module}$), a sort ($\mathcal{A}_{sort}$), a symbol ($\mathcal{A}_{symbol}$), a cell ($\mathcal{A}_{cell}$), a rewriting rule ($\mathcal{A}_{rule}$), a context ($\mathcal{A}_{context}$) or a context alias ($\mathcal{A}_{context-alias}$). Moreover, the configuration variable begins by `$` such as `$PGM`, the fresh variable begins by `!` and the unbound variable in rewriting rule begins by `?`. Finally, there are three cast operators noted `:`, `::` and `:>`.

The $\mathbb{K}$ grammar on Figure 6 is almost complete. In order to not make it too heavy, we have omitted a part of the syntax which allows to declare the precedence, the associativity and the priority of symbols since this is only useful for the generation of the $\mathbb{K}$ parser extended with the user-defined language $\mathcal{L}$ (Figure 5).

In addition, we consider that the attributes related to symbolic execution ($\mathcal{A}_{\text{KProver}}$) have been deleted as well as the following syntactic sugar:
`syntax` **BExp** `::=` **MyBool** `|` **BExp** `"&&"` **BExp** is syntactic sugar for
`syntax` **BExp** `::=` **MyBool**
`syntax` **BExp** `::=` **BExp** `"&&"` **BExp**.
We do not take into account either the syntax `...` and the fact that the rewriting arrow `=>` can be nested since Li and Gunter [16] explain that this syntax is ambiguous syntactic sugar. We assume that all these syntactic sugars are deleted by the black box "Desugar" (Figure 5).

In this article, we only consider a $\mathbb{K}$ semantics if $\mathbb{K}$ accepts it as well as the $\mathbb{K}$ grammar (Figure 6) – after deleting a part of the syntax and some syntactic sugars.

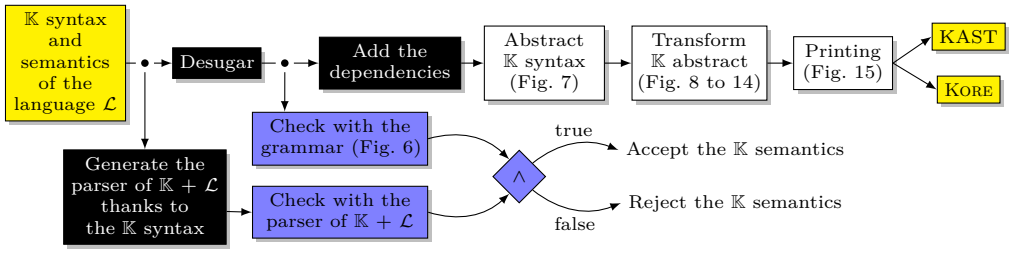

Figure 5 Overview of the translation from $\mathbb{K}$ semantics, under condition

We just explained the two first black boxes on Figure 5. Other boxes are explained (black one) or formalized (white one) in the next section.

3 Abstracting the $\mathbb{K}$ framework

This section presents a mathematical structure $\mathcal{M}$ to abstract the syntax of $\mathbb{K}$. After the presentation of the structure $\mathcal{M}$ as well as the translation of the $\mathbb{K}$ syntax into the structure $\mathcal{M}$, we present various transformations of the structure $\mathcal{M}$ which correspond to the static semantics of $\mathbb{K}$. To present this work, we start with the output of the black box "Add the dependencies" (Figure 5) which recursively replaces each `require` command with the contents of these files, and then recursively replaces each `import` command with the contents of the module that must be imported. We consider that the output of the black box is a single module whose name corresponds to the name of the main initial semantics file. At this stage, the following attributes have finished influencing the transformation: `hook`, $\mathcal{A}_{visibility}$, $\mathcal{A}_{backend}$, `not-lr1`, $\mathcal{A}_{parsing}$, `token`, $\mathcal{A}_{token}$, `locations`, $\mathcal{A}_{printing}$ and $\mathcal{A}_{\text{KProver}}$.

```
⟨carac⟩              ::=  [ a-zA-Z | 0-9 | - | _ ]
⟨int⟩                ::=  [ 1-9 ][ 0-9 ]*
⟨string⟩             ::=  " ⟨carac⟩* "
⟨name-module⟩        ::=  ⟨carac⟩+
⟨symbol⟩             ::=  ⟨carac⟩+
⟨str-of-reg-expr⟩    ::=  a regular expression between quotes

⟨require⟩            ::=  ( require | requires ) " ⟨carac⟩+ [ .k | .md ] "
⟨import⟩             ::=  ( import | imports ) [ public | private ]? ⟨name-module⟩

⟨sort⟩               ::=  [A-Z #] ⟨carac⟩*
⟨sort-syntax⟩        ::=  syntax ⟨sort⟩ ( [ token | locations | A_sort ,+ ] )?
                     |    syntax ⟨sort⟩ ::= ⟨sort⟩ ( [ token ] )?

⟨terminal⟩           ::=  ⟨string⟩
⟨non-terminal⟩       ::=  ⟨sort⟩
⟨syntax-item⟩        ::=  ⟨terminal⟩ | ⟨non-terminal⟩
⟨separator⟩          ::=  ⟨string⟩
⟨syntax⟩             ::=  syntax [ { ⟨sort⟩,+ } ]? ⟨sort⟩ ::= ⟨symbol⟩( ⟨sort⟩,* ) [ A_symbol ,* ]
                     |    syntax [ { ⟨sort⟩,+ } ]? ⟨sort⟩ ::= ⟨syntax-item⟩+ [ A_symbol ,* ]
                     |    syntax ⟨sort⟩ ::= r ⟨str-of-reg-expr⟩ [ { token } ∪ A_token ,* ]
                     |    syntax ⟨sort⟩ ::= List   { ⟨sort⟩, ⟨separator⟩ }
                     |    syntax ⟨sort⟩ ::= NeList { ⟨sort⟩, ⟨separator⟩ }

⟨config-variable⟩    ::=  $[A-Z]+
⟨initial-value⟩      ::=  ⟨symbol⟩ | ⟨config-variable⟩
⟨cell⟩               ::=  < ⟨name⟩ A_cell ,*> ⟨initial-value⟩ [ : ⟨sort⟩ ]? < / ⟨name⟩>
                     |    < ⟨name⟩ A_cell ,*> ⟨cell⟩+ < / ⟨name⟩>
⟨configuration⟩      ::=  configuration ⟨cell⟩+

⟨variable⟩           ::=  [ ? | ! ]? [A-Z] ⟨carac⟩* | _
⟨pattern⟩            ::=  ( ⟨variable⟩ | ⟨symbol⟩ ) [ : ⟨sort⟩ | :: ⟨sort⟩ ]?
                     |    { ⟨pattern⟩+ } :> ⟨sort⟩
⟨rule⟩               ::=  rule ⟨pattern⟩+ => ⟨pattern⟩+ [ requires ⟨pattern⟩+ ]? [ A_rule ,* ]

⟨context⟩            ::=  context ⟨pattern⟩+ [ requires ⟨pattern⟩+ ]? ([ A_context ,+ ])?
⟨context-alias⟩      ::=  context alias [ ⟨carac⟩+ ]: ⟨pattern⟩+ ([ A_context−alias ,+ ])?

⟨sentence⟩           ::=  ⟨sort-syntax⟩    | ⟨syntax⟩
                     |    ⟨configuration⟩ | ⟨rule⟩
                     |    ⟨context⟩ | ⟨context-alias⟩

⟨module⟩             ::=  module ⟨name-module⟩ ([ A_module ,+ ])?
                               ⟨import⟩*
                               ⟨sentence⟩*
                          endmodule

⟨file⟩               ::=  ⟨require⟩* ⟨module⟩*

where  A_module      ≜  A_visibility ∪ A_backend ∪ { not-lr1 }
       A_symbol      ≜  A_parsing ∪ A_family ∪ A_property ∪ A_strategy ∪ A_modulo ∪ A_printing ∪ A_visibility
       A_cell        ≜  A_structure ∪ A_console ∪ { unused="", color= ⟨string⟩ }
       A_context     ≜  { result }
       A_context−alias ≜ { result, context }
```

Figure 6 The considered $\mathbb{K}$ grammar, where $\boxed{X}$ is any subset of X

3.1 An abstract view of $\mathbb{K}$

To abstract a $\mathbb{K}$ file, we use the 7-uplet $\mathcal{M} \triangleq (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias})$, where $\mathcal{S}ort$ is the set of sorts, $\mathcal{R}el$ is the set of subtyping relations, $\mathcal{S}ym$ is the set of symbols, $\mathcal{C}onfig$ is the set of configurations, $\mathcal{R}$ is the set of rewriting rules, $\mathcal{C}ontext$ is the set of contexts and $\mathcal{C}ontext_{alias}$ is the set of contexts alias. Figure 7 shows the translation of the $\mathbb{K}$ syntax to the abstract structure $\mathcal{M}$. Syntactic declarations can create sorts, relations between two sorts, or symbols. As $\mathbb{K}$ allows mixfix notations, we translate them into prefix notations such as || `syntax` Exp `::=` "if" `Bool` "then" Exp "else" Exp || = || `syntax` Exp `::=` if-then-else`(`Bool, Exp, Exp`)` ||. Moreover, a configuration declaration generates a list of trees l. Section 3.2 shows that l is transformed into a single tree. Finally, any unconditional rewriting rules can be seen as a conditional rule with the condition "true".

|| `syntax` s (`[` $Attr$ `]`)? || $= \mathcal{S}ort \leftarrow \{ s \}$
|| `syntax` s_1 `::=` s_2 (`[` $Attr$ `]`)? || $= \mathcal{S}ort \leftarrow \{ s_1 ; s_2 \} ; \mathcal{R}el \leftarrow \{ s_2 < s_1 \}$
|| `syntax` `{` α_1, ..., α_n `}` α `::=` sym `(` s_1, ..., s_x `)` `[` $Attr$ `]` || $= (n \geq 0$ and $x \geq 0)$
$\mathcal{S}ort \leftarrow \{ \alpha ; s_1 ; ... ; s_x \} \setminus \{ \alpha_1, ..., \alpha_n \} ; \mathcal{S}ym \leftarrow \{ sym : \forall\alpha_1, ..., \forall\alpha_n, s_1 \rightarrow ... \rightarrow s_x \rightarrow \alpha\ [Attr] \}$
|| `syntax` `{` α_1, ..., α_n `}` α `::=` i_1 ... i_x `[` $Attr$ `]` || $= (n \geq 0$ and $x \geq 1)$
|| `syntax` `{` α_1, ..., α_n `}` α `::=` t_1-...-t_n `(` s_1, ..., s_x `)` `[` $Attr$ `]` ||
where $t_i \in I_{terminal} \triangleq \{ i_k \mid k \in [\![1; x]\!]$ and $i_k \in \langle$**terminal**$\rangle \}$
$s_i \in I_{non-terminal} \triangleq \{ i_k \mid k \in [\![1; x]\!]$ and $i_k \in \langle$**non-terminal**$\rangle \}$

|| `syntax` s `::=` `r` $\langle$**str-of-reg-expr**$\rangle$ `[` $Attr$ `]` || $= \mathcal{S}ort \leftarrow \{ s \}$
|| `syntax` s_1 `::=` `List` `{` s_2`,` sep `}` || $= \mathcal{S}ort \leftarrow \{ s_1 ; s_2 \}$
|| `syntax` s_1 `::=` `NeList` `{` s_2`,` sep `}` || $= \mathcal{S}ort \leftarrow \{ s_1 ; s_2 \}$

|| `configuration` $cell_1$ ... $cell_n$ || $= \mathcal{C}onfig \leftarrow (||\ cell_1\ ||_{\texttt{rec}} ; ... ; ||\ cell_n\ ||_{\texttt{rec}})$, with $n \geq 1$
|| `<` C `>` v `:` s `</` C `>` $||_{\texttt{rec}}$ $= [C]_s(v)$ (If s isn't given, we can infer it from v.)
|| `<` C `>` $cell_1$ ... $cell_n$ `</` C `>` $||_{\texttt{rec}} = <C>(||\ cell_1\ ||_{\texttt{rec}}, ..., ||\ cell_n\ ||_{\texttt{rec}})$

|| `rule` LHS `=>` RHS `[` $Attr$ `]` || $= \mathcal{R} \leftarrow (LHS \overset{\texttt{true}}{\hookrightarrow} RHS\ [Attr])$
|| `rule` LHS `=>` RHS `requires` $Cond$ `[` $Attr$ `]` || $= \mathcal{R} \leftarrow (LHS \overset{Cond}{\hookrightarrow} RHS\ [Attr])$

|| `context` C (`[` $Attr$ `]`)? || $= \mathcal{C}ontext \leftarrow (C, \texttt{true}, Attr)$
|| `context` C `requires` $Cond$ (`[` $Attr$ `]`)? || $= \mathcal{C}ontext \leftarrow (C, Cond, Attr)$
|| `context` `alias` `[` $label$ `]:` CA (`[` $Attr$ `]`)? || $= \mathcal{C}ontext_{alias} \leftarrow (label, CA, Attr)$

Figure 7 From $\mathbb{K}$ syntax to 7-uplet, where $X \leftarrow data$ is the set X with the new element $data$

From the obtained mathematical structure $\mathcal{M}$, the following subsection explains the internal transformations made by $\mathbb{K}$.

3.2 Compilation of a $\mathbb{K}$ semantics

This subsection formalizes the transformations on the previous mathematical structure $\mathcal{M}$ which correspond to the static semantics of $\mathbb{K}$. We assume that initially $\mathcal{S}ort = \mathcal{S}ort_{lib}$, $\mathcal{R}el = \mathcal{R}el_{lib}$, $\mathcal{S}ym = \mathcal{S}ym_{lib}$ and $\mathcal{R} = \mathcal{R}_{lib}$ whereas $\mathcal{C}onfig$, $\mathcal{C}ontext$ and $\mathcal{C}ontext_{alias}$ are empty. The first transformation (Figure 8) has for goal to check the validity of a $\mathbb{K}$ semantics.

Each symbol should be a *constructor*, a *function*, a *token*, a *bracket* or a *macro*. Regarding the attribute `macro`, the documentation is not precise enough. According to Li and Gunter [15], macros are subject to many errors when writing semantics, but in practice macros are always considered as syntactic sugar. The associated rewriting rules are used only once at the beginning of an evaluation of the input programs, to rewrite the syntactic sugar into another term. Macros can therefore be replaced by functions as they are not more expressive.

Moreover, a semantics must have only one configuration, and some attributes have precise restrictions.

$$
\begin{array}{l}
||\ (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias})\ ||_{\texttt{check-input}} = \\
\quad (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym', \mathcal{C}onfig', \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias}) \\
\quad\quad \text{where} \\
\quad\quad\quad \begin{array}{lcl}
\mathcal{C} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{constructor} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{F} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{function} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{T} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{token} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{B} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{bracket} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{M} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{macro} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{W} & \triangleq & \mathcal{S}ym \setminus (\mathcal{C} \cup \mathcal{F} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{M}) \\
\mathcal{W}' & \triangleq & \{\ n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [\mathcal{A}ttr \cup \{\texttt{constructor}\}] \mid\ n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [\mathcal{A}ttr] \in \mathcal{W}\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{C} \cup \mathcal{F} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{M} \cup \mathcal{W}' \\
\mathcal{C}onfig' & \triangleq & [k]_{\texttt{K}}(\texttt{\$PGM}) \text{ if } \mathcal{C}onfig = \emptyset;\ \mathcal{C}onfig \text{ if } \mathrm{card}(\mathcal{C}onfig) = 1
\end{array} \\
\quad \textbf{Constraints:} \\
\quad\quad \text{The set } \{\ \mathcal{C},\ \mathcal{F},\ \mathcal{T},\ \mathcal{B},\ \mathcal{M},\ \mathcal{W}\ \} \text{ must be a partition of } \mathcal{S}ym. \\
\quad\quad \mathcal{C}onfig \text{ is the empty set or a singleton, where each cell have a unique name.} \\
\quad\quad \text{For all } n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [\mathcal{A}ttr] \in \mathcal{F},\ \{\ \texttt{strict}, \texttt{seqstrict}\ \} \cap \mathcal{A}ttr = \emptyset. \\
\quad\quad \text{For all } n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [\mathcal{A}ttr] \in \mathcal{B},\ \overrightarrow{v} = \overrightarrow{0} \text{ and } \overrightarrow{t} = x \text{ and } x = \alpha.
\end{array}
$$

Figure 8 Formalization of the function $||\ .\ ||_{\texttt{check-input}}$

Generate evaluation strategy rules thanks to contexts and contexts alias.

We have seen that to define evaluation strategies in $\mathbb{K}$, we can define contexts alias, contexts or use the attributes `strict` and `seqstrict`. We assume the existence of a function $||\ .\ ||_{\texttt{delete-context-alias}}$ that transforms a context alias into contexts as well as a function $||\ .\ ||_{\texttt{delete-context}}$ that transforms a context into rewriting rules. The lack of information in the documentation prevents us from formalising these two functions.

Generate evaluation strategy rules thanks to attributes.

We formalize on Figure 9 the rule generation from the attributes `strict` and `seqstrict`.

$$
\begin{array}{l}
||\ (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R})\ ||_{\texttt{generate-strategy}} = (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym', \mathcal{C}onfig, \mathcal{R}') \\
\quad \text{where} \\
\quad\quad \begin{array}{lcl}
\mathcal{S}_{\texttt{strict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{strict} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{S}_{\texttt{seqstrict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{seqstrict} \in \mathcal{A}ttr(s)\ \} \\
\mathcal{F}reezer & \triangleq & \{\ \circledast^{n_j}_{sym} : \texttt{K} \to \ldots \to \texttt{K} \to \texttt{KItem}\ [\texttt{constructor}\ ] \\
 & & \quad \mid \text{with nbArg}(sym) \text{ - } k \text{ argument(s)} \\
 & & \quad\quad \text{and } \texttt{strict}\ (n_1,\ldots,n_k) \in \mathcal{A}ttr(sym) \\
 & & \quad\quad \text{or } \texttt{seqstrict}\ (n_1,\ldots,n_k) \in \mathcal{A}ttr(sym)\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{S}ym \cup \mathcal{F}reezer \\
\mathcal{R}' & \triangleq & \mathcal{R} \cup \mathcal{R}_{\texttt{strict}} \cup \mathcal{R}_{\texttt{seqstrict}}
\end{array}
\end{array}
$$

$\mathcal{R}_{\texttt{strict}}$ is composed of the following rewriting rules:
For all $sym \in \mathcal{S}_{\texttt{strict}}$ such that $\texttt{strict}\ (n_1,\ldots,n_j) \in \mathcal{A}ttr(sym)$, with $k \in \{\ n_1, \ldots, n_j\ \}$:
$sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\circledast^k_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ]$, where $c \triangleq \neg\ (\texttt{isKResult}\ E_k)$
$E_k \curvearrowright (\circledast^k_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ]$, where $c \triangleq \texttt{isKResult}\ E_k$

$\mathcal{R}_{\texttt{seqstrict}}$ is composed of the following rewriting rules:
For all $sym \in \mathcal{S}_{\texttt{seqstrict}}$ such that $\texttt{seqstrict}\ (n_1,\ldots,n_j) \in \mathcal{A}ttr(sym)$, with $k \in \{\ n_1, \ldots, n_j\ \}$:
$sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\circledast^k_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ]$
where $c \triangleq \texttt{isKResult}\ E_1 \wedge \ldots \wedge \texttt{isKResult}\ E_{k-1} \wedge \neg\ (\texttt{isKResult}\ E_k)$
$E_k \curvearrowright (\circledast^k_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ]$, where $c \triangleq \texttt{isKResult}\ E_k$

Constraint:
Every argument of the attribute `strict` or `seqstrict` should be in $[1; arity(sym)]$.

Figure 9 Formalization of the function $||\ .\ ||_{\texttt{generate-strategy}}$

For example, the rules 1. and 2. (Figure 3) illustrated the formalization on Figure 9.

Encapsulate the configuration.

According to the grammar on Figure 6, a configuration is a list of finite branching trees. But $\mathbb{K}$ encapsulates any configuration $[cell_1; ...; cell_n]$ as follows, where GT $\triangleq$ GeneratedTop and GC $\triangleq$ GeneratedCounter: <GT>(<T>($cell_1$, ..., $cell_n$), $[\text{GC}]_{\texttt{Int}}(0)$). For instance, the initial configuration from Figure 2 becomes $\langle\ \langle\ \langle$ `$PGM` : **BExp** $\rangle_k\ \rangle_T\ \langle$ 0 $\rangle_{GC}\ \rangle_{GT}$.
Thus, after this transformation, any configuration becomes a single finite branching tree.

Generate implicit cells.

Now that we have the entire configuration, we formalize the generation of all the implicit cells in the rewriting rules (Figure 10).

$$
\begin{array}{lll}
\multicolumn{3}{l}{|| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}) ||_{\texttt{add-cell}} = (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}')} \\
\text{where} & & \\
 & \mathcal{R}' \triangleq & \{\ || \ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ and } x \triangleq \text{mgconf } \mathcal{C}onfig\ \} \\
\text{with} & \text{mgconf } (<C>(Cell_1, ..., Cell_n)) \triangleq & <C>(\text{mgconf } Cell_1, ..., \text{mgconf } Cell_n) \\
 & \text{mgconf } ([C]_s(v)) \triangleq & [C]_s(f), \text{ where } f \text{ is a fresh variable} \\
\text{with} & || \ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x \triangleq & l \overset{c}{\hookrightarrow} r\ [Attr], \text{ if the heading symbol of } l \text{ is a function symbol} \\
 & & l \overset{c}{\hookrightarrow} r\ [Attr], \text{ if } \texttt{anywhere} \in Attr \\
 & & \overline{l}^x \overset{c}{\hookrightarrow} \overline{r}^x\ [Attr], \text{ otherwise} \\
\text{with} & \overline{p}^x \triangleq x\ [\ \overrightarrow{[C]_s(y)} \setminus \overrightarrow{[C]_s(v)}\ ] & \text{where } \overrightarrow{[C]_s(v)} \text{ are the leaves appearing in } p
\end{array}
$$

Figure 10 Formalization of the function $||\ .\ ||_{\texttt{add-cell}}$

Thus, mgconf $\mathcal{C}onfig$ is $\langle\ \langle\ \langle\ pgm\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$, where $\mathcal{C}onfig$ is the initial configuration of Figure 2, pgm and c are fresh variables. Moreover, the rule 4. from Figure 3 becomes: `rule` $\langle\ \langle\ \langle$ `false && _` $\curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle$ `false` $\curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$.

Split the configuration.

Now we are ready to decompose the configuration into sorts and symbols. The generated rewriting rules are useful to complete the initial configuration with the value given by the user thanks to the configuration variable. The formalization is available on Figure 11, where GT $\triangleq$ GeneratedTop and GC $\triangleq$ GeneratedCounter.

$$
\begin{array}{lll}
\multicolumn{3}{l}{|| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}) ||_{\texttt{split-config}} = (\mathcal{S}ort', \mathcal{R}el, \mathcal{S}ym', \mathcal{R}')} \\
\text{where} & & \\
 & \mathcal{S}ort' \triangleq & \mathcal{S}ort \cup \{\ \text{Sort}C \mid <C>(c_1, ..., c_n) \in \mathcal{C}onfig\ \} \\
 & \mathcal{S}ym_{cell} \triangleq & \{\ C : \text{Type } (\ <C>(c_1, ..., c_n)\ )\ [\texttt{constructor}\ ] \mid <C>(c_1, ..., c_n) \in \mathcal{C}onfig\} \\
 & \mathcal{S}ym_{init} \triangleq & \{\ \text{init}C : \text{Sort}C\ [\texttt{function}, \texttt{initializer}\ ] \mid <C>(c_1, ..., c_n) \in \mathcal{C}onfig\} \\
 & \mathcal{S}ym_{get} \triangleq & \{\ \text{getGCCell} : \text{SortGTCell} \to \text{SortGCCell}\ [\texttt{function}\ ]\ \} \\
 & \mathcal{S}ym' \triangleq & \mathcal{S}ym \cup \mathcal{S}ym_{cell} \cup \mathcal{S}ym_{init} \cup \mathcal{S}ym_{get} \\
 & \mathcal{R}_{init} \triangleq & \{\ \text{init}C \hookrightarrow <C>((\text{GetInit } c_1), ..., (\text{GetInit } c_n))\ [\texttt{initializer}\ ] \\
 & & \qquad \mid <C>(c_1, ..., c_n) \in \mathcal{C}onfig\} \\
 & \mathcal{R}' \triangleq & \mathcal{R} \cup \mathcal{R}_{init} \cup \{\ \text{getGCCell}(<\text{GT}>(\text{X},\text{V})) \hookrightarrow \text{V}\ \} \\
\text{with} & \text{Type } (\ <X>(c_1, ..., c_n)\ ) \triangleq & \text{RetType}(c_1) \to ... \to \text{RetType}(c_n) \to \text{Sort}X \\
\text{with} & \text{RetType } (\ <X>(c_1, ..., c_n)\ ) \triangleq & \text{Sort}X \\
 & \text{RetType } (\ [X]_s(v)\ ) \triangleq & s \\
\text{with} & \text{GetInit } (\ <X>(c_1, ..., c_n)\ ) \triangleq & \text{init}X \\
 & \text{GetInit } (\ [X]_s(v)\ ) \triangleq & v
\end{array}
$$

Figure 11 Formalization of the function $||\ .\ ||_{\texttt{split-config}}$

For example, we obtained the symbol `initK : SortK` and the rule `initK` $\hookrightarrow$ `$PGM` as well as the symbols `GT : SortT → SortGC → SortGT` and `T : SortK → SortT`.

Add implicit attributes.

Implicitly, every symbol with the attribute `constructor` has also the attributes `total`, formerly called `functional`, and `injective`, as formalized on Figure 12.

$$
\begin{array}{l}
\| \ (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \ \|_{\texttt{add-attributes}} = (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym', \mathcal{R}) \\
\quad \text{where} \\
\qquad \begin{array}{lcl}
\mathcal{S}_{\texttt{constructor}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \text{if } \texttt{constructor} \in Attr(s)\ \} \\
\mathcal{S}'_{\texttt{constructor}} & \triangleq & \{\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr \cup \{\texttt{total}, \texttt{injective}\}] \\
 & & \qquad \mid\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{S}_{\texttt{constructor}}\ \} \\
\mathcal{S}ym' & \triangleq & (\mathcal{S}ym \setminus \mathcal{S}_{\texttt{constructor}}) \ \cup \ \mathcal{S}'_{\texttt{constructor}}
\end{array}
\end{array}
$$

Figure 12 Formalization of the function $\| \ . \ \|_{\texttt{add-attributes}}$

Manage the fresh values.

Any fresh variable can be replaced by the value presents on the cell `GeneratedCounterCell`. For instance, `rule` $\langle\ \langle\ \langle\ \texttt{foo X} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle\ \texttt{!X} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ becomes `rule` $\langle\ \langle\ \langle\ \texttt{foo X} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle\ c \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c + 1\ \rangle_{GC}\ \rangle_{GT}$.

Add the typing symbols and extend the typing hierarchy.

Implicitly, every user-defined sort is a subsort of `KItem`. Moreover, $\mathbb{K}$ generates projection symbols and predicate symbols such as `isKResult`, as shown on Figure 13.

$$
\begin{array}{l}
\| \ (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \ \|_{\texttt{typing}} = (\mathcal{S}ort, \mathcal{R}el', \mathcal{S}ym', \mathcal{R}') \\
\quad \text{where} \\
\qquad \begin{array}{lcl}
\mathcal{R}el' & = & \mathcal{R}el \ \cup \ \{s < \texttt{KItem} \mid s \in \mathcal{S}ort\} \\
\mathcal{F}_{projection} & = & \{\ \texttt{proj}s : \texttt{K} \to s\ [\texttt{projection}, \texttt{function}\ ] \mid s \in \mathcal{S}ort\ \} \\
\mathcal{F}_{predicate} & = & \{\ \texttt{is}s : \texttt{K} \to \texttt{Bool}\ [\texttt{predicate}, \texttt{function}, \texttt{functional}\ ] \mid s \in \mathcal{S}ort\ \} \\
\mathcal{S}ym' & = & \mathcal{S}ym \ \cup \ \mathcal{F}_{projection} \ \cup \ \mathcal{F}_{predicate} \\
\mathcal{R}_{projection} & = & \{\ s\ (\texttt{inj}_t^{\texttt{KItem}}\ X) \hookrightarrow X\ [\texttt{projection}\ ] \\
 & & \qquad \mid s \in \mathcal{F}_{projection} \text{ if } t \text{ is the output type of } s\ \} \\
\mathcal{R}_{predicate} & = & \{\ s\ (\texttt{inj}_t^{\texttt{KItem}}\ X) \hookrightarrow \texttt{true}\ [\ ] \mid s \in \mathcal{F}_{predicate} \text{ if } t \text{ is the output type of } s\ \} \\
\mathcal{R}_{pred-owise} & = & \{\ s\ X \hookrightarrow \texttt{false}\ [\texttt{owise}\ ] \mid s \in \mathcal{F}_{predicate}\ \} \\
\mathcal{R}' & = & \mathcal{R} \ \cup \ \mathcal{R}_{projection} \ \cup \ \mathcal{R}_{predicate} \ \cup \ \mathcal{R}_{pred-owise}
\end{array}
\end{array}
$$

Figure 13 Formalization of the function $\| \ . \ \|_{\texttt{typing}}$

Checking the coherence of the typing hierarchy.

We can construct a graph where the nodes are elements of $\mathcal{S}ort$ and the edges are modelled by the elements of $\mathcal{R}el$. We reject the semantics if the graph contains at least one cycle.

Add injections.

$\mathbb{K}$ adds also injections (`inj`) to get full well-typed terms. This step takes into account also the constraints of semantic casts (`:`), strict casts (`::`) and projection casts (`:>`).

Checking the constraints on the attribute `binder`.

The first argument of a symbol with the attribute `binder` must have the type `KVar`, which is a native $\mathbb{K}$ sort. Then, to do a substitution, it is the responsibility of each backend to correctly implement the interface proposed by $\mathbb{K}$, i.e. the file `substitution.md`.

Checking the constraints on `List` and `NeList`.

It is not possible to add two or more `List` or/and `NeList` constructions at the same time to the same sort. So it is not allowed to write `syntax` **Exp** `::= List{Int,","} | List{Bool,","}`.

Checking the constraints on the $\mathbb{K}$ standard library.

The user cannot extend the $\mathbb{K}$ standard library with `constructor` symbols, so the set $\{\ n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{S}ym \mid \text{if } \texttt{constructor} \in Attr \text{ and } \alpha \in \mathcal{S}ort_{lib}\ \}$ should be empty. That's the reason we named a sort **MyBool** and not **Bool** on Figure 2.

Checking the constraints on rewriting rules.

Finally, each rewriting rule needs to respect the BNF on Figure 14 and the type of every condition must be boolean.

$$
\begin{array}{lll}
a & ::= & x \mid (\sigma\ a\ \ldots\ a) \\
b & ::= & x \mid (\sigma\ b\ \ldots\ b) \mid (f\ b\ \ldots\ b) \\
c & ::= & f\ b\ \ldots\ b \\
rule & ::= & \sigma\ a\ \ldots\ a \hookrightarrow b \mid \sigma\ a\ \ldots\ a \overset{c}{\hookrightarrow} b \mid f\ a\ \ldots\ a \hookrightarrow b \mid f\ a\ \ldots\ a \overset{c}{\hookrightarrow} b \\
\\
\text{where} & \sigma \text{ is a } \texttt{constructor} \text{ symbol} & \\
& f \text{ is a } \texttt{function} \text{ symbol} & \\
& x \text{ is a variable} &
\end{array}
$$

Figure 14 Constraints on rewriting rules

We have just presented the various translations carried out internally by $\mathbb{K}$. We do not claim that this list is exhaustive but it reflects our understanding of the translation from $\mathbb{K}$ to Kore. This paper formalization was elaborated by reverse engineering on Kore files as well as thanks to discussions with the $\mathbb{K}$ teams. It seems possible to print a (almost always valid) new $\mathbb{K}$ file after each transformation but this has not been implemented. Only the following attributes have not yet had an influence to the transformation: $\mathcal{A}_{modulo}$, { `priority()`, `owise` }, { `multiplicity`, `type`, `exit`, `stream` } and { `injective`, `total` }.

3.3 From $\mathbb{K}$ to Kore

We present the translation from the abstraction of $\mathbb{K}$ into Kore (Figure 15). Thanks to the obtained quadruplet, we can translate a $\mathbb{K}$ semantics into a specific Matching Logic theory named Kore. Every red keyword can be translated into a Matching Logic pattern but this translation is beyond the scope of this article. The pattern φ_{sym} can take 3 different forms which correspond to the axioms of injectivity, non-overlap and exhaustivity of constructors.

$$
\begin{array}{ll}
\| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \|_{\text{Kore}} = & \\
\texttt{sort}\ s\{\alpha_1,\ldots,\alpha_n\}\ [\ ] & \text{for all } s \in \mathcal{S}ort \setminus \mathcal{S}ort_{lib} \\
\texttt{hooked-sort}\ s\{\alpha_1,\ldots,\alpha_n\}\ [\ ] & \text{for all } s \in \mathcal{S}ort_{lib} \\
\texttt{symbol}\ sym\{\alpha_1,\ldots,\alpha_n\}(\theta_1,\ldots,\theta_m) : \theta'\ [\ \text{Attr}\ ] & \text{for all } sym \in \mathcal{S}ym \setminus \mathcal{S}ym_{lib} \\
\texttt{hooked-symbol}\ sym\{\alpha_1,\ldots,\alpha_n\}(\theta_1,\ldots,\theta_m) : \theta'\ [\ \text{Attr}\ ] & \text{for all } sym \in \mathcal{S}ym_{lib} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash exists}\ \{\text{R}\}(x_2 : \theta_2, & \\
\quad \texttt{\textbackslash equals}\ \{\theta_2, \text{R}\}(x_2 : \theta_2, \texttt{inj}\ \{\theta_1, \theta_2\}(x_1 : \theta_1)))\ [\ \texttt{subsort}\ (\theta_1, \theta_2)\ ] & \text{for all } \theta_1 < \theta_2 \in \mathcal{R}el \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash exists}\ \{\text{R}\}(x : \theta, \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(x : \theta, sym\ x_1\ \ldots\ x_n))\ [\ \texttt{total}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{total}} \\
\texttt{axiom}\ \{\alpha_1,\ldots,\alpha_n\}\ \varphi_{sym}\ [\ \texttt{constructor}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{constructor}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(\ sym(sym(x_1 : \theta, x_2 : \theta), x_3 : \theta), & \\
\qquad\qquad sym(x_1 : \theta, sym(x_2 : \theta, x_3 : \theta)))\ [\ \texttt{assoc}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{assoc}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(sym(x_1 : \theta, x_2 : \theta), sym(x_2 : \theta, x_1 : \theta))\ [\ \texttt{comm}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{comm}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(sym(e, x : \theta), x : \theta)\ [\ \texttt{unit}\ ] & \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(sym(x : \theta, e), x : \theta)\ [\ \texttt{unit}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{unit}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash equals}\ \{\theta, \text{R}\}(sym(x : \theta, x : \theta), x : \theta)\ [\ \texttt{idem}\ ] & \text{for all } sym \in \mathcal{S}_{\texttt{idem}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash implies}\ \{\text{R}\}(c, \texttt{\textbackslash equals}\ \{\text{R}, \text{R}\}(l, r))\ [\ Attr\ ] & \text{for all } l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{function}} \\
\texttt{axiom}\ \{\text{R}\}\ \texttt{\textbackslash rewrites}\ \{\text{R}\}(\texttt{\textbackslash and}\ \{\text{R}\}(c, l), r)\ [\ Attr\ ] & \text{for all } l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{constructor}} \\
\multicolumn{2}{l}{\quad \text{where } \mathcal{S}_a \triangleq \{\ s \in \mathcal{S}ym \mid \text{if } a \in Attr(s)\ \} \text{ and } \mathcal{R}_a \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \mid \text{if } a \in Attr(head(l))\ \}}
\end{array}
$$

Figure 15 The printer to Kore

4 From the $\mathbb{K}$ framework to the $\lambda\Pi$-calculus modulo theory

This section presents $\lambda\Pi$-calculus modulo theory and Dedukti, a logical framework based on it. Then, we formalize the translation from $\mathbb{K}$ to $\lambda\Pi$-calculus modulo theory. This formalization is implemented as a tool written in OCaml, named KaMeLo, which is presented in the next section.

4.1 $\lambda\Pi$-calculus modulo theory

The $\lambda\Pi$-calculus modulo theory, $\lambda\Pi\equiv_{\mathcal{T}}$ in short, is a logical framework, i.e. allowing to define theories, introduced by Cousineau and Dowek [11]. $\lambda\Pi\equiv_{\mathcal{T}}$ is an extension of the λ-calculus with dependent types and a primitive notion of computation defined thanks to rewriting rules [12]. The syntax as well as the typing rules that define the $\lambda\Pi\equiv_{\mathcal{T}}$ are available on Figure 16, where the typing judgment $\Gamma \vdash t : A$ means that the term t has type A under the context Γ. The specific typing judgment $\Gamma \vdash A : \mathbf{Type}$ indicates that A is a type under the context Γ. We also consider a signature Σ, and a set of higher-order rewriting rules $\mathcal{R}$. In this framework, any rewriting rule $l \hookrightarrow r \in \mathcal{R}$ verifies $FV(r) \subseteq FV(l)$, where $FV(p)$ is the free variables of p, and the use of such a rewriting rule requires that any instantiations $l\sigma$ and $r\sigma$ of its left- and right-hand sides are well-typed with the same type ($\Gamma \vdash l\sigma : A$ and $\Gamma \vdash r\sigma : A$ for a certain type A).

Syntax	s	$:=$	$\mathbf{Type} \mid \mathbf{Kind}$	*sort*
	t	$:=$	$s \mid c \mid x \mid t\ t \mid \lambda(x:t).t \mid \Pi(x:t).t$	*term*
	Γ	$:=$	$\emptyset \mid \Gamma, x : t$	*context*
		where	c is a constant of Σ,	
			x is a variable	

Typing

$$(\text{sort})\ \frac{}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}}$$

$$(\text{const})\ \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash c : A}\ (c : A) \in \Sigma \qquad (\text{var})\ \frac{\Gamma \vdash A : \mathbf{Type}}{\Gamma \vdash x : A}\ (x : A) \in \Gamma$$

$$(\text{app})\ \frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \backslash a\}} \qquad (\text{abs})\ \frac{\Gamma \vdash \Pi(x : A).B : s \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \Pi(x : A).B}$$

$$(\text{prod})\ \frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi(x : A).B : s} \qquad (\text{conv})\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \equiv_{\beta\mathcal{R}} B$$

$$(\equiv_{reduc})\ \frac{}{\Gamma \vdash (\lambda(x : A).t)\ u \equiv t\{x \backslash u\}} \qquad (\equiv_{rule})\ \frac{\Gamma \vdash l\sigma : A \qquad \Gamma \vdash r\sigma : A}{\Gamma \vdash l\sigma \equiv r\sigma}\ l \hookrightarrow r \in \mathcal{R}$$

where $s \in \{\mathbf{Type}\ ; \mathbf{Kind}\}$, $B\{x \backslash a\}$ is the substitution of a for x in B, and $\equiv_{\beta\mathcal{R}}$ is the reflexive, transitive, symmetric and contextual closure of $\equiv$, generated by the rules $\equiv_{reduc}$ and $\equiv_{rule}$.

Figure 16 Syntax and typing of $\lambda\Pi\equiv_{\mathcal{T}}$ with a signature Σ and a rewriting system $\mathcal{R}$

Note that in the conversion rule (conv), the equivalence relation depends not only on the β-reduction but also on the rewriting system $\mathcal{R}$. Moreover, in order to have the decidability of the typing, the condition $A \equiv_{\beta\mathcal{R}} B$ in the rule (conv) must be decidable, which is ensured when the considered rewriting systems are convergent. Finally, contexts can contain ill-formed elements and the order of the elements does not matter. However, thanks to the rule (var), only well-formed elements in the context can be used when doing a proof. This presentation has been shown to be equivalent to the usual presentations in Dowek [13].

4.2 Dedukti

Dedukti [6, 1, 2] is a logical framework based on the $\lambda\Pi\equiv_{\mathcal{T}}$. Indeed, expressing the Calculus of Constructions in Dedukti is equivalent to defining it as a theory of the $\lambda\Pi\equiv_{\mathcal{T}}$. Several logics have been encoded in the common language Dedukti, which facilitates the interoperability of proofs such as the reuse of proofs from Coq in PVS. In this section, we only present the features available in Dedukti needed in this article.

4.2.1 Typing and symbols

The syntax of the $\lambda\Pi\equiv_{\mathcal{T}}$ is directly accessible in Dedukti: `TYPE` (**Kind**, not being accessible to the user but only inferred by the system), λ (abstraction), Π (dependent product). When the dependent product $\Pi\ (x : A), B$ is in fact not dependent, i.e. when $x \notin FV(B)$, we write $A \to B$.

The signature is defined from symbols. If the declaration of a symbol is made with the keyword `symbol` alone, the symbol is said to be *defined*, without any particular property, whereas with the additional keyword `constant`, the symbol is said to be *constant* and can't be reduced by a rewriting rule.

4.2.2 Rewriting rules

A Dedukti rule is written `rule` $LHS \hookrightarrow RHS$ in which the free variables are noted `$x`, `$y`, etc. As $\mathbb{K}$, it is possible to use a wildcard (`_`) on the left-hand side when a free variable is not used in the right-hand side. Dedukti rules allow higher-order, can be non-linear and do not necessarily apply to the head of the term, but are not conditional.

4.3 Translation from $\mathbb{K}$ abstract to $\lambda\Pi$-calculus modulo theory

Section 3 presented an abstract version of $\mathbb{K}$: any $\mathbb{K}$ file can thus be reduced to a set of sorts, subtyping relations, symbols and rewriting rules. Figure 17 presents the translation of these sets into $\lambda\Pi$-calculus modulo theory.

$$
\begin{array}{l}
|| \ (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \ ||_{\lambda\Pi\equiv_{\mathcal{T}}} = \\
\quad \texttt{K} : \textbf{Type} \\
\quad s : \texttt{K} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for all } s \in \mathcal{S}ort \setminus \{\texttt{K}\} \\
\quad n : \Pi(a_1 : \alpha_1), ..., \Pi(a_n : \alpha_n), \Pi(t_1 : \tau_1), ..., \Pi(t_n : \tau_n), \alpha \qquad \text{for all } n : \forall\overrightarrow{\alpha}, \overrightarrow{\tau} \to \alpha \ [Attr] \in \mathcal{S}ym \\
\quad l \hookrightarrow r \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional} \\
\quad || \ \mathcal{R}_{conditional} \ ||_{\texttt{CTRS}} \\
\quad || \ (\mathcal{R}el, \ \mathcal{S}ym, \ \mathcal{R}_{strategy}) \ ||_{\texttt{strategy}} \\
\\
\quad \text{where } \mathcal{R}_{unconditional} \triangleq \{ \ l \hookrightarrow r \mid l \overset{c}{\hookrightarrow} r \ [Attr] \in \mathcal{R} \text{ if } c = \texttt{true} \ \} \\
\quad \text{where } \mathcal{R}_{conditional} \triangleq \{ \ l \overset{c}{\hookrightarrow} r \ [Attr] \mid l \overset{c}{\hookrightarrow} r \ [Attr] \in \mathcal{R} \text{ if } c \neq \texttt{true} \text{ and } \{ \ \texttt{heat}, \texttt{cool} \ \} \cap Attr = \emptyset \ \} \\
\quad \text{where } \mathcal{R}_{strategy} \triangleq \{ \ l \overset{c}{\hookrightarrow} r \ [Attr] \mid l \overset{c}{\hookrightarrow} r \ [Attr] \in \mathcal{R} \text{ if } \texttt{heat} \in Attr \text{ or } \texttt{cool} \in Attr \ \}
\end{array}
$$

Figure 17 From $\mathbb{K}$ to $\lambda\Pi\equiv_{\mathcal{T}}$

Any sort becomes a symbol of type `K`, except the sort `K` itself, which has the type **Type**. Symbols and unconditional rules are unchanged and the set of rewriting rules obtained at the end of the translation $||\ .\ ||_{\lambda\Pi\equiv_{\mathcal{T}}}$ is $\{ \ l \hookrightarrow r \mid \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional} \ \} \cup || \ \mathcal{R}_{conditional} \ ||_{\texttt{CTRS}} \cup || \ (\mathcal{R}el, \ \mathcal{S}ym, \ \mathcal{R}_{strategy}) \ ||_{\texttt{strategy}}$.

The translation function $||\ .\ ||_{\texttt{CTRS}}$ is explained on Section 4.3.1 and the translation function $||\ .\ ||_{\texttt{strategy}}$ is explained on Section 4.3.2.

4.3.1 Translating conditional rewriting rules

In this section, we are interested in the translation of conditional rewriting rules. As it is not possible to define conditional rewriting rules in $\lambda\Pi\equiv_{\mathcal{T}}$, it is necessary to find an encoding of a conditional rewriting system ($CTRS$) into a non-conditional rewriting system (TRS).

4.3.1.1 From a CTRS to a TRS: Examples

We present two examples to illustre the encoding of a CTRS into a TRS.

An example without `owise`. Consider the following system:
(1) $max\ X\ Y \overset{c}{\hookrightarrow} Y$, where $c \triangleq X < Y$
(2) $max\ X\ Y \overset{c}{\hookrightarrow} X$, where $c \triangleq X \geq Y$.
The resulting encoding is available on Figure 18 as well as an execution.

(0)	$max\ X\ Y \hookrightarrow \flat max\ X\ Y\ \flat\ \flat$	$max\ 5\ 3 \hookrightarrow_0 \flat max\ 5\ 3\ \flat\ \flat$
(1′)	$\flat max\ X\ Y\ \flat\ C \hookrightarrow \flat max\ X\ Y\ (X < Y)\ C$	$\hookrightarrow_{1'} \flat max\ 5\ 3\ (5 < 3)\ \flat$
(1″)	$\flat max\ X\ Y\ true\ C \hookrightarrow Y$	$\hookrightarrow^* \flat max\ 5\ 3\ false\ \flat$
(2′)	$\flat max\ X\ Y\ C\ \flat \hookrightarrow \flat max\ X\ Y\ C\ (X \geq Y)$	$\hookrightarrow_{2'} \flat max\ 5\ 3\ false\ (5 \geq 3)$
(2″)	$\flat max\ X\ Y\ C\ true \hookrightarrow X$	$\hookrightarrow^* \flat max\ 5\ 3\ false\ true \hookrightarrow_{2''} 5$

Figure 18 Rules generated with the variant of Viry's encoding (left) and an execution (right)

The general idea of the encoding, proposed in this section and initially proposed by Viry [24], is to add as many arguments as there are conditions for a symbol defined with conditional rules. The rule (0) rewrites a term whose heading symbol is max to a term using the corresponding extended version of arity 4, $\flat max$ here, where all boolean arguments are $\flat$, which indicate that the boolean arguments have not yet been initialised by a condition. Rules (1′) and (2′) initialize the conditions to be computed whereas rules (1″) and (2″) reduce the size of the term since one of the conditions has been evaluated to $true$. This encoding has the advantage of not fixing the order of evaluation of the conditions but increases the computation time by doubling the initial number of rules.

Contrary to Viry, we choose to extend the signature, as here with the symbol $\flat max$, rather than replacing each symbol of the signature by an equivalent symbol but with a greater arity. This choice makes it possible to follow the computations of the conditions and does not force us to translate the obtained normal forms.

An example with `owise`. The previous example can also be written more succinctly:
$max\ X\ Y \overset{c}{\hookrightarrow} Y$, where $c \triangleq X < Y$
$max\ X\ Y \hookrightarrow X$ [`owise`]
To encode the attribute `owise`, we have two possibilities: to implement an algorithm which determines the complementary condition or to consider that all conditions necessarily reduce to either `true` or `false`. According to the expressiveness of the conditions which can be written in $\mathbb{K}$ (Figure 14), we add the following hypothesis: any function which constructs a boolean is a total function. Under this assumption, it is not required to compute the complementary condition as we can generate a rule where every boolean argument is $false$, as shown on Figure 19. This case is formalized in 5.(c) (Figure 20).

(0)	$max\ X\ Y \hookrightarrow \flat max\ X\ Y\ \flat$
(1′)	$\flat max\ X\ Y\ \flat \hookrightarrow \flat max\ X\ Y\ (X < Y)$
(1″)	$\flat max\ X\ Y\ true \hookrightarrow Y$
(2′)	$\flat max\ X\ Y\ false \hookrightarrow X$

Figure 19 Rules generated with the variant of Viry's encoding, when the attribute `owise` is used

Furthermore, $\mathbb{K}$ accepts a set of unconditional rules with at least one rule having the attribute `owise`. We exclude this case because we cannot model "If no other rule applies", with a boolean condition. This is equivalent to the use of the attribute `priority(nb)`, which we do not yet deal with.

4.3.1.2 From a CTRS to a TRS: Formalization

We present the translation, noted $||\ .\ ||_{\texttt{CTRS}}$ previously, as an algorithm on Figure 20. This translation takes as argument a set $\mathfrak{R}$ of conditional rewriting rules $l \overset{c}{\hookrightarrow} r\ [Attr]$. To avoid naming conflicts, we also assume that $\flat$ is an unused symbol name and that it does not appear in the head of any symbol name.

According to Figure 10, we need to change the definition of the *heading symbol of a rule*. We note $head_{\texttt{<k>}}$ the function which returns the heading symbol of the cell `<k>`, for a given rule, without considering the symbols `.`, $\curvearrowright$ and `inj`. Now, we are able to define the function which returns the heading symbol of a rule:

$$head(l \overset{c}{\hookrightarrow} r\ [Attr]) = \begin{cases} head_{\texttt{<k>}}(l \overset{c}{\hookrightarrow} r) & \text{if the rule is a semantical rule and } \texttt{anywhere} \notin Attr \\ f & \text{otherwise, where } l \triangleq f\ a\ ...\ a \text{ (Figure 14)} \end{cases}$$

We also define the set of rules noted $\mathcal{C}_\sigma$, which the rules share the same heading symbol σ, that is $\mathcal{C}_\sigma \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid head\ (l \overset{c}{\hookrightarrow} r\ [Attr]) = \sigma\ \}$.

After constructing each set $\mathcal{C}_\sigma$ from $\mathfrak{R}$, we run the algorithm presented on Figure 20, for each $\mathcal{C}_\sigma$, where X is the number of conditional rules in $\mathcal{C}_\sigma$.

1. If $X = 0$, $\mathcal{C}_\sigma$ is unchanged and the algorithm stops. Otherwise, initialize i to 0 and go to 2.
2. Generate the most general left-hand side for a given symbol σ, noted $mglhs_\sigma$.
3. Generate the extended symbol $\flat\sigma$ of type $T_1 \to ... \to T_{n-1} \to \flat\texttt{Bool} \to ... \to \flat\texttt{Bool} \to T_n$, with X argument(s) of type $\flat\texttt{Bool}$, where $\flat\texttt{Bool} = \texttt{Bool} \cup \{\flat\}$, and σ of type $T_1 \to ... \to T_n$.
4. Generate the substitution rule: $mglhs_\sigma \hookrightarrow mglhs_\sigma[\ update_{same}(\sigma, \flat\sigma, \flat)\ ]_\sigma$
5. For each rule $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{C}_\sigma$:
   - **a.** If $c \neq \texttt{true}$ and $\texttt{owise} \notin Attr$:
     - Increment i by 1
     - Generate the initialization rule: $l\ [\ update_{diff}(\sigma, \flat\sigma, \flat, i, \_)\ ]_\sigma \hookrightarrow l\ [\ update_{diff}(\sigma, \flat\sigma, c, i, \_)\ ]_\sigma$
     - Generate the reduction rule: $l\ [\ update_{diff}(\sigma, \flat\sigma, \texttt{true}, i, \_)\ ]_\sigma \hookrightarrow r$
   - **b.** If $c = \texttt{true}$ and $\texttt{owise} \notin Attr$:
     - Generate the reduction rule: $l\ [\ update_{same}(\sigma, \flat\sigma, \_)\ ]_\sigma \hookrightarrow r$
   - **c.** If $c = \texttt{true}$ and $\texttt{owise} \in Attr$:
     - Generate the reduction rule: $l\ [\ update_{same}(\sigma, \flat\sigma, \texttt{false})\ ]_\sigma \hookrightarrow r$

where:
* $t_1[t_2]_\sigma$ means that we substitute t_2 for the subterm with the heading symbol σ in t_1.
* $arg_i(t)$ corresponds to the i-th argument of t and $arity(t)$ to the number of arguments of t.
* $mglhs_\sigma \triangleq$ (mgconf *init-config*) [$[\![\texttt{k}]\!]_\texttt{K}(x) \setminus [\![\texttt{k}]\!]_\texttt{K}((\texttt{inj}^{\texttt{KItem}}_{RetType(\sigma)}\ \sigma\ f_1\ ...\ f_{arity(\sigma)}) \curvearrowright L)$], where *init-config* is the initial configuration, and f_i and L are fresh variables.
* $update_{diff}(\sigma, \flat\sigma, s_1, i, s_2) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ *with* $x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \leq j \leq arity(\sigma) \\ s_1 & \text{if } j = arity(\sigma) + i \\ s_2 & \text{otherwise} \end{cases}$
* $update_{same}(\sigma, \flat\sigma, s) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ *with* $x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \leq j \leq arity(\sigma) \\ s & \text{otherwise} \end{cases}$

Figure 20 Variant of Viry's encoding

4.3.2 Translating evaluation strategies

As we saw in Section 3.2, some conditional rewriting rules can be generated during the translation of $\mathbb{K}$ to Kore, as is the case for the evaluation strategies defined by the attributes `strict` and `seqstrict`. The rewriting rules generated by these attributes require the translation of the K computations, i.e. the symbols $.$ and $\curvearrowright$, the freezers but also the predicate `isKResult`. However, these conditional rewriting rules are part of a known case where Viry's encoding is not confluent, notably because the order of application of some rewriting rules modifies the result of the condition, which can block the computation. Indeed, Figure 21 shows the translation of the rules of Figure 3 with Viry's encoding (on the right) and an example of a valid but blocked execution[1] (on the left).

1	$E1$ `&&` $E2 \curvearrowright C \hookrightarrow$ $\flat$`&&` $E1$ $E2$ $\flat \curvearrowright C$	$(true$ `&&` $true)$ `&&` $false \curvearrowright .$ $\hookrightarrow_1 \flat$`&&` $(true$ `&&` $true)$ $false$ $\flat \curvearrowright .$
2	$\flat$`&&` $E1$ $E2$ $\flat \curvearrowright C \hookrightarrow$ $\flat$`&&` $E1$ $E2$ $(not($`isKResult` $E1)) \curvearrowright C$	$\hookrightarrow_2 \flat$`&&` $(true$ `&&` $true)$ $false$ $(not($`isKResult` $(true$ `&&` $true))) \curvearrowright .$
3	$\flat$`&&` $E1$ $E2$ $true \curvearrowright C \hookrightarrow E1 \curvearrowright (\clubsuit^1_{\&\&}$ E2$) \curvearrowright C$	
4	$E1 \curvearrowright (\clubsuit^1_{\&\&}$ E2$) \curvearrowright C \hookrightarrow$ $(\flat \clubsuit^1_{\&\&}$ E1 E2 $\flat) \curvearrowright C$	$\hookrightarrow^* \flat$`&&` $(true$ `&&` $true)$ $false$ $true \curvearrowright .$ $\hookrightarrow_3 (true$ `&&` $true) \curvearrowright (\clubsuit^1_{\&\&} false) \curvearrowright .$
5	$(\flat \clubsuit^1_{\&\&}$ E1 E2 $\flat) \curvearrowright C \hookrightarrow$ $(\flat \clubsuit^1_{\&\&}$ E1 E2 $($`isKResult` E1$)) \curvearrowright C$	$\hookrightarrow_4 (\flat \clubsuit^1_{\&\&} (true$ `&&` $true)$ $false$ $\flat) \curvearrowright .$ $\hookrightarrow_5 (\flat \clubsuit^1_{\&\&} (true$ `&&` $true)$
6	$(\flat \clubsuit^1_{\&\&}$ E1 E2 $true) \curvearrowright C \hookrightarrow E1$ `&&` $E2 \curvearrowright C$	$false$ $($`isKResult` $(true$ `&&` $true))) \curvearrowright .$
7	$true$ `&&` $B \curvearrowright C \hookrightarrow B \curvearrowright C$	
8	$false$ `&&` $\_ \curvearrowright C \hookrightarrow false \curvearrowright C$	$\hookrightarrow^* (\flat \clubsuit^1_{\&\&} (true$ `&&` $true)$ $false$ $false) \curvearrowright .$

Figure 21 Rules generated with previous encoding (left) and a blocked execution (right)

Intuitively, these rules are used to ensure that E_1 is of a specific type in order to allow or not its evaluation. The idea of our new encoding is to specialize some terms, as E_1, of the rules, i.e. to refine the pattern-matching to ensure the desired type. As example, the rule 2. on Figure 3 becomes $\langle\ (\mathtt{inj}^{\mathtt{KItem}}_{\mathtt{Bool}}\ E_1) \curvearrowright (\clubsuit^1_{\&\&}\ E_2) \curvearrowright S\ \rangle_k \hookrightarrow \langle\ (\mathtt{inj}^{BExp}_{\mathtt{Bool}}\ E_1)$ `&&` $E_2 \curvearrowright S\ \rangle_k$, where we force E_1 to have the type `Bool`. Thus this rule can be used only if the term E_1 is a fully computed Boolean expression. The rule 1. on Figure 3 becomes only the rule $\langle\ (X_1$ `&&` $X_2)$ `&&` $E_2 \curvearrowright S\ \rangle_k \hookrightarrow \langle\ (X_1$ `&&` $X_2) \curvearrowright (\clubsuit^1_{\&\&}\ E_2) \curvearrowright S\ \rangle_k$ because there is only one constructor associated to **BExp** and no `token` symbol. Symbols with the attribute `function` or `macro` are not considered, because they are not allowed in the left-hand side of a rule, as well as with the attribute `bracket`, because they disappear during the compilation process of $\mathbb{K}$. The full formalization is available on Figure 22.

$$
\begin{array}{l}
\| (\mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \|_{\mathtt{strategy}} = \mathcal{R}' \\
\quad \text{where} \quad \mathcal{S}ub \triangleq \{\ s \mid s < \mathtt{KResult} \in \mathcal{R}el\ \} \\
\qquad \mathcal{R}_{\mathtt{cool}} \triangleq \{\ r \in \mathcal{R} \mid \mathtt{cool} \in Attr(r)\ \} \\
\qquad \mathcal{R}'_{\mathtt{cool}} \triangleq \{\ (l \hookrightarrow r)\ [\ x \setminus \mathtt{inj}^{\mathtt{KItem}}_{s}\ x\ ] \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\mathtt{cool}} \text{ where } c \triangleq (\mathtt{isKResult}\ x),\ s \in \mathcal{S}ub\ \} \\
\qquad \mathcal{R}_{\mathtt{heat}} \triangleq \{\ r \in \mathcal{R} \mid \mathtt{heat} \in Attr(r)\ \} \\
\qquad \mathcal{S}^{s_2}_{s_1} \triangleq \{\ \mathtt{inj}^{s_2}_{s}\ f \mid \text{where } f \text{ is a fresh variable and } s \in (\{\ s \mid s < s_1 \in \mathcal{R}el\} \setminus \mathcal{S}ub)\ \} \\
\qquad \mathcal{P}^{s_2}_{s_1} \triangleq \{\ \mathtt{inj}^{s_2}_{s_1}(n\ \overrightarrow{f}) \mid \text{where } \overrightarrow{f} \text{ are fresh variables and } n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{S}ym \\
\qquad\qquad \text{if } \alpha = s_1 \text{ and } \{\ \mathtt{constructor}, \mathtt{token}\ \} \cap Attr \neq \emptyset\ \} \\
\qquad \mathcal{R}'_{\mathtt{heat}} \triangleq \{\ (l \hookrightarrow r)\ [\ x_1, \ldots, x_k \setminus \mathtt{inj}^{\mathtt{KItem}}_{s_1}\ x_1, \ldots, \mathtt{inj}^{\mathtt{KItem}}_{s_k}\ x_k\ ]\ [\ \mathtt{inj}^{s_2}_{s_1}\ x \setminus t\ ] \\
\qquad\qquad \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\mathtt{heat}} \text{ and } (s_1, \ldots, s_k) \in \mathcal{S}ub^k \text{ and } t \in (\mathcal{S}^{s_2}_{s_1} \cup \mathcal{P}^{s_2}_{s_1}), \\
\qquad\qquad\quad \text{where } c \triangleq (\mathtt{isKResult}\ x_1 \wedge \ldots \wedge \mathtt{isKResult}\ x_k \wedge \neg\ (\mathtt{isKResult}\ \mathtt{inj}^{s_2}_{s_1}\ x)\ \} \\
\qquad \mathcal{R}' \triangleq \mathcal{R} \setminus (\mathcal{R}_{\mathtt{heat}} \cup \mathcal{R}_{\mathtt{cool}}) \cup (\mathcal{R}'_{\mathtt{heat}} \cup \mathcal{R}'_{\mathtt{cool}})
\end{array}
$$

Figure 22 Specialization of the evaluation strategy rules

[1] It is also possible to obtain `false`.

4.3.3 Semantics preservation

The correctness of the translation is not formally proved in this article. Informally, our translation seeks to ensure that the program executed in the $\mathbb{K}$ framework and the program executed in Dedukti have the same behaviour. If the language described is deterministic, $\mathbb{K}$ and Dedukti compute the same value or give the same final state. If a language is non-deterministic, $\mathbb{K}$ allows to obtain all possible final configurations. In Dedukti, it is only possible to obtain one final configuration, because the algorithm is deterministic.

As previously, we assume that every condition is reducible into `false` or `true`. We also assume that the $\mathbb{K}$ semantics does not use the following attributes: no cell of the configuration has one of the attributes `multiplicity`, `stream`, `type`, `exit`, no evaluation strategy based on `result` or `hybrid`, and no rewriting rule has the attribute `priority()`, `unboundVariables`, `assoc`, `comm`, `unit` or `idem`. Lastly, we assume that, for a given symbol, among the associated computational rules, only one rule has the attribute `owise`.

The two following parts present the expected results, thanks to the function $|\ .\ |$ which is defined inductively on Term($\mathbb{K}$):

- $|\ sym\ x_1\ ...\ x_n\ | \triangleq sym\ |\ x_1\ |\ ...\ |\ x_n\ |$, if $sym \in \Sigma_{\text{Dedukti}}$,
- $|\ x\ | \triangleq x$, where x is a variable

4.3.3.1 Correctness

▶ **Conjecture 1.** *For every rewriting step $l \hookrightarrow r$ in $\mathbb{K}$, there is a derivation $|\ l\ | \hookrightarrow^* |\ r\ |$ in Dedukti.*

▶ **Corollary** (From $\mathbb{K}$ to Dedukti)**.** *For every derivation $l \hookrightarrow^* r$ in $\mathbb{K}$, there is a derivation $|\ l\ | \hookrightarrow^* |\ r\ |$ in Dedukti.*

4.3.3.2 Completeness

We note $\mathcal{F}lat$ the set of every term which starts by $\flat$.
We note $\mathcal{G}host$ the set of every term which has at least one symbol in $\mathcal{F}lat$.

▶ **Lemma 1.** *$|\ .\ |$: Term($\mathbb{K}$) $\rightarrow$ Term(Dedukti) $\setminus\ \mathcal{G}host$ is a bijection.*

We define the *translation function* $||\ .\ ||_{\texttt{K2DK}}$: Term($\mathbb{K}$) $\rightarrow$ Term(Dedukti) such that $||\ t\ ||_{\texttt{K2DK}} = |\ t\ |$ and the *detranslation function* $||\ .\ ||_{\texttt{DK2K}}$: Term(Dedukti) $\rightarrow$ Term($\mathbb{K}$) such that $||\ t\ ||_{\texttt{DK2K}} = \begin{cases} |\ t\ |^{-1} & \text{if } t \notin \mathcal{G}host \\ ||\ t\ ||_{\texttt{forget}} & \text{if } t \in \mathcal{G}host \end{cases}$.
The function `forget` is defined inductively on Term(Dedukti):

- $||\ sym\ x_1\ ...\ x_n\ ||_{\texttt{forget}} \triangleq sym\ ||\ x_1\ ||_{\texttt{forget}}\ ...\ ||\ x_n\ ||_{\texttt{forget}}$, if $sym \notin \mathcal{F}lat$
- $||\ \flat sym\ x_1\ ...\ x_n\ ||_{\texttt{forget}} \triangleq sym\ ||\ x_1\ ||_{\texttt{forget}}\ ...\ ||\ x_i\ ||_{\texttt{forget}}$,
  if $\flat sym \in \mathcal{F}lat$, where $x_{i+1}\ ...\ x_n$ are conditions
- $||\ x\ ||_{\texttt{forget}} \triangleq x$, where x is a variable

▶ **Conjecture 2** (From Dedukti to $\mathbb{K}$)**.** *For every derivation $l \hookrightarrow^* r$ in Dedukti, there is a derivation $||\ l\ ||_{DK2K} \hookrightarrow^* ||\ r\ ||_{DK2K}$ in $\mathbb{K}$ if $l \notin \mathcal{G}host$ and $r \notin \mathcal{G}host$.*

▶ **Corollary** (Preservation of confluence)**.** *If the rewriting system $\mathcal{R}$ written in $\mathbb{K}$ is confluent, then the translation of the rewriting system $\mathcal{R}$ in Dedukti is confluent.*

▶ **Corollary** (Preservation of termination)**.** *If the rewriting system $\mathcal{R}$ written in $\mathbb{K}$ is terminating, then the translation of the rewriting system $\mathcal{R}$ in Dedukti is terminating.*

The next section presents the implementation of the translations presented in this section.

5 Implementation and examples

This section focuses on the tool KaMeLo [3], which allows to translate Kore into Dedukti.

5.1 KaMeLo in a nutshell

In practice, the formalizations presented in Section 3.2 as well as the printer to Kore (Figure 15) correspond to the command `kompile` implemented by the 𝕂 team. Like the `krun` command, also implemented by the 𝕂 team, KaMeLo allows programs translated into Kore to be executed in Dedukti thanks to the 𝕂 semantics translated into Kore. KaMeLo implements the translation formalized on Figure 20 and Figure 22.

Moreover, it is the responsibility of each backend to implement the 𝕂 standard library and to support the appropriate attributes. The backend KaMeLo does not support the attributes `multiplicity`, `stream`, `type`, `exit`, `result`, `hybrid`, `priority()`, `unboundVariables`, `assoc`, `comm`, `unit` and `idem`. The implementation of the 𝕂 standard library in Dedukti is available on `https://gitlab.com/semantiko/DK-BiblioteKo`.

5.2 KaMeLo in action

From a semantics of 84 lines of an imperative language, it is possible to obtain a Kore file of 4 130 lines (18 sorts, 5 hooked sorts, 102 symbols, 78 hooked symbols and 552 axioms.). However, in order to execute a program, the axioms with the attributes `subsort`, `total`, `constructor`, `assoc`, `comm`, `unit` or `idem` do not need to be translated. The translation of this semantics in Dedukti has 723 lines (122 symbols and 122 rewriting rules).
To execute the following program which computes the PGCD of x and y

```
decl x, y ; x = 20 ; y = 15 ;
while not( (y <= x) and (x <= y) ) do {
  if y < x
  then x = x - y ;
  else y = y - x ;
}
```

the command `$ krun -depth 0 -output kore PGCD.imp` allows to translate the program in Kore. After translating it into Dedukti, the result is: `<generatedTop> (<T> (<k> .) (<env> (inj y ↦ inj 5) ; (inj x ↦ inj 5))) (<generatedCounter> 0);`.
The source code of KaMeLo [3] is joined by some tests as the one presented here.

6 Conclusion

This article presents a paper formalization of the translation from 𝕂 semantics into Kore and, a paper formalization and an automatic tool, called KaMeLo, from Kore to Dedukti, in order to execute programs in Dedukti. There has already been a translation of a programming language in Dedukti [9, 10], but this is the first time a semantical framework has been translated into Dedukti.

This work needs to be extended to take into account the attributes `priority()`/`owise`, `multiplicity`/`type` and `result`/`hybrid`. The attributes `assoc`, `comm`, `unit`, `idem` and `unboundVariables` can theoretically not be translated in the general case.

The verification of proof objects generated by the KProver as well as the encoding of the theoretical foundations of 𝕂 into those of Dedukti, are not in the scope of this article and will be the subject of future work. The translation presented here is nevertheless necessary to run a program and will be reused for proof checking.

References

1 GitHub of Dedukti. https://github.com/Deducteam/Dedukti.
2 GitHub of Dedukti v3. https://github.com/Deducteam/lambdapi.
3 GitLab of KaMeLo. https://gitlab.com/semantiko/kamelo.
4 Website of $\mathbb{K}$. https://kframework.org/.
5 Website of Sail. https://www.cl.cam.ac.uk/~pes20/sail/.
6 Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the $\lambda\Pi$-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016. URL: https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751.
7 Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. doi:http://dx.doi.org/10.1145/2676726.2676982.
8 P. Borras, D. Clement, T. Despeyroux, Janet Bertot, G. Kahn, Bernard Lang, and Valérie Pascual. Centaur: The system. 24:14–24, 03 1989. doi:10.1145/64140.65005.
9 Raphaël Cauderlier and Catherine Dubois. ML Pattern-Matching, Recursion, and Rewriting: From FoCaLiZe to Dedukti. In *ICTAC 2016 - 13th International Colloquium on Theoretical Aspects of Computing*, volume 9965 of *LNCS*, pages 459–468, Taipei, Taiwan, October 2016. URL: https://hal.inria.fr/hal-01420638, doi:10.1007/978-3-319-46750-4_26.
10 Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the rescue for proof interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017: International Conference on Interactive Theorem Proving*, number 10499, page 532, Brasília, Brazil, September 2017. URL: https://hal.inria.fr/hal-01592243, doi:10.1007/978-3-319-66107-0_9.
11 D. Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.
12 Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320, 1990.
13 Gilles Dowek. Interacting Safely with an Unsafe Environment. *CoRR*, abs/2107.07662, 2021. URL: https://arxiv.org/abs/2107.07662, arXiv:2107.07662.
14 Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015. doi:http://dx.doi.org/10.1145/2813885.2737979.
15 Liyi Li and Elsa L. Gunter. IsaK: A Complete Semantics of K. Technical report, June 2018. URL: https://hdl.handle.net/2142/100116.
16 Liyi Li and Elsa L. Gunter. IsaK-static: A complete static semantics of K. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings*, pages 196–215. Springer-Verlag Berlin Heidelberg, 2018. URL: https://experts.illinois.edu/en/publications/isak-static-a-complete-static-semantics-of-k, doi:10.1007/978-3-030-02146-7_10.
17 Liyi Li and Elsa L. Gunter. A Complete Semantics of $\mathbb{K}$ and Its Translation to Isabelle. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021*, pages 152–171, Cham, 2021. Springer International Publishing.
18 Dominic Mulligan, Scott Owens, Kathryn Gray, Tom Ridge, and Peter Sewell. Lem: Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, 49, 08 2014. doi:10.1145/2628136.2628143.
19 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015. doi:http://dx.doi.org/10.1145/2737924.2737991.

20 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, August 2010. URL: https://www.sciencedirect.com/science/article/pii/S1567832610000160, doi:10.1016/j.jlap.2010.03.012.

21 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. doi:10.1017/S0956796809990293.

22 Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91, Amsterdam Netherlands, October 2016. ACM. URL: https://dl.acm.org/doi/10.1145/2983990.2984027, doi:10.1145/2983990.2984027.

23 M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment: A Component-Based Language Development Environment. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. doi:10.1007/3-540-45306-7_26.

24 Patrick Viry. Elimination of Conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999. URL: https://www.sciencedirect.com/science/article/pii/S0747717199902882, doi:https://doi.org/10.1006/jsco.1999.0288.

25 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, nov 1994. doi:10.1006/inco.1994.1093.