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, Samovar, Évry-Courcouronnes, 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 their properties thanks to the provided automatic theorem prover called the KProver. $\mathbb{K}$ is based on Matching Logic, a first-order logic with an application and fixed-point operators, 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 $\mathbb{K}$ 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.2022.12

Supplementary Material *Software*: `https://gitlab.com/semantiko/kamelo`
archived at `swh:1:dir:9353026d393b52e5bc149a469a3d2386fb923dff`

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. Before verifying a program, it must be written in a programming language whose syntax and semantics are precisely known. 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 for example Centaur [7], ASF+SDF [23], Ott [20], Sail [3], Lem [17] or $\mathbb{K}$ [19, 2]. 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 [6], C [13] or JavaScript [18].

$\mathbb{K}$ is a semantical framework that offers many features 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 offers the possibility of verifying some properties – expressed in the form of reachability properties – on the program $\mathcal{P}$ using the automatic theorem prover KProver [21]. 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 is the reason we look at the translation of $\mathbb{K}$ into Dedukti, which is a logical framework based on the $\lambda\Pi$-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 into Dedukti of any semantics written in $\mathbb{K}$ in order to execute programs in Dedukti. Our first contribution is a paper formalization of the translation from $\mathbb{K}$ 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 [15, 16], 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 [1], from Kore to Dedukti in order to execute programs in Dedukti. The general overview of the translation pipeline presented in this article is available in Figure 1. This is the first translation into Dedukti involving a semantical framework.

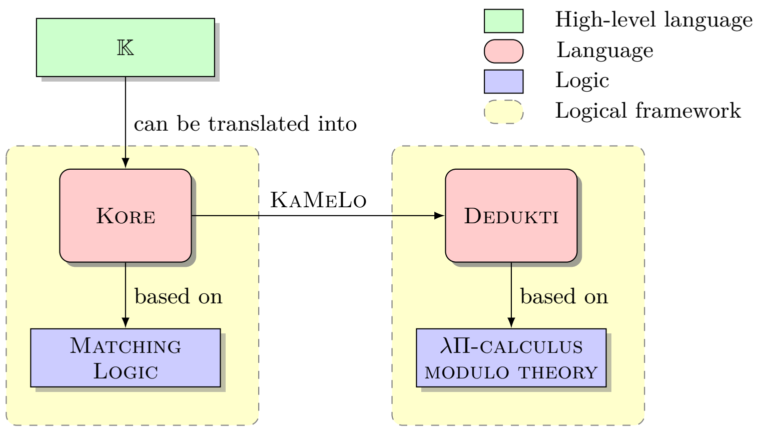


Figure 1 Overview of the translation pipeline presented in this article.

A long-term goal is not only to execute a program $\mathcal{P}$ written with the formalized language $\mathcal{L}$ within Dedukti, but also to 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 about 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 semantics (Section 3). Thanks to the mathematical structure $\mathcal{M}$, we formalize a computational translation $\mathcal{T}$ into the $\lambda\Pi$-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 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 𝕂 framework?

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

2.1 A first 𝕂 semantics

In this subsection, we show an example based on booleans and two binary symbols, which are lazily evaluated, to illustrate how to write a semantics in 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 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, which are typed by sort **MyBool**, a lazy disjunction, noted `||`, and a lazy conjunction, noted `&&`. 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 the other possible attributes as we go along. Moreover, 𝕂 supports the Kleene operators “*” and “+”, written `List` and `NeList` respectively.

```
 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 | module LAZY
11 |      imports LAZY-SYNTAX
12 |      configuration <k> $PGM : BExp </k>
   |
13 |      rule false || B => B
14 |      rule true  || _ => true
   |
15 |      rule true  && B => B
16 |      rule false && _ => false
17 | endmodule
```

Figure 2 Syntax and semantics of booleans, a lazy disjunction and a lazy conjunction.

The subtyping relation between the sorts **MyBool** and **BExp** (Figure 2 - line 5) means that the boolean values `true` and `false` are also boolean expressions. In addition, any symbol has either the attribute `constructor`, when the symbol is an element of the syntax, or the attribute `function`, when the symbol is a *helper function* used to define the semantics, e.g. functions to manipulate the environment.

2.1.2 Define the semantics associated to the syntax

The main ingredients for defining the semantics of each element of the syntax are *configurations* and *rewriting rules*. Some attributes are also useful to define evaluation strategies. The semantics of a lazy disjunction and a lazy conjunction is defined in the module LAZY (Figure 2 - lines 10 to 17) which imports the syntax module (line 11 thanks to `imports`).

2.1.2.1 Configurations

A *configuration* models the state of the program and is composed of *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 and the other labelled by *env* containing the current values of the variables. In the example in Figure 2, the configuration contains only the cell k (line 12). The configuration variable `$PGM` will contain the parsed program given by the user.

2.1.2.2 Rewriting rules

A 𝕂 rewriting rule is a 1st order rule which can be either conditional, noted `rule` *LHS* `=>` *RHS* `requires` *Cond*, or unconditional, noted `rule` *LHS* `=>` *RHS*. Moreover, a 𝕂 rewriting rule can be non-linear, i.e. variables in the left-hand side can appear several times. 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 14 or 16 (Figure 2). Finally, 𝕂 supports partial rewriting modulo ACUI, i.e. associativity (`assoc`), commutativity (`comm`), identity (`unit`) and idempotence (`idem`).

Any 𝕂 rewriting rule can be applied to a whole configuration, if the rewriting rule defines the semantics associated to the syntax, or does not mention the configuration, if the rewriting rule defines a helper function. This distinction is 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 aliases* (`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 15 or 16. To use these attributes, the user needs to define the sort `KResult` which allows to distinguish final values from expressions thanks to subtyping. For instance, as **MyBool** is a sub-sort of `KResult` (Figure 2 - line 4), 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 list of computations to be performed sequentially and built with the constructors `.` and $\curvearrowright$, whereas a freezer is a symbol that encapsulates the part of the computation that should not 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 [26] and 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

$(\circledast^{nb}_{sym}\ \text{arg})$ 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 15 and 16 (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`, so $\mathbb{K}$ does not transform the rules on lines 13 and 14 (Figure 2). The translation of the attribute `strict(1)` into rewriting rules is similar in the case of attributes `strict` and `seqstrict`.

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

$$\begin{array}{lcl}
\langle\, (\texttt{true \&\& false})\ \texttt{\&\&}\ (\texttt{true \&\& true}) \curvearrowright \texttt{.}\,\rangle_k & & \langle\, e_1\ \texttt{\&\&}\ e_2 \curvearrowright s\,\rangle_k \\
\hookrightarrow_1 \langle\, (\texttt{true \&\& false}) \curvearrowright (\circledast^1_{\texttt{\&\&}}\ (\texttt{true \&\& true})) \curvearrowright \texttt{.}\,\rangle_k & & \hookrightarrow_1 \langle\, e_1 \curvearrowright (\circledast^1_{\texttt{\&\&}}\ e_2) \curvearrowright s\,\rangle_k \\
\quad\hookrightarrow_3 \langle\, \texttt{false} \curvearrowright (\circledast^1_{\texttt{\&\&}}\ (\texttt{true \&\& true})) \curvearrowright \texttt{.}\,\rangle_k & \text{abstracted by} & \quad\hookrightarrow_3 \langle\, v_1 \curvearrowright (\circledast^1_{\texttt{\&\&}}\ e_2) \curvearrowright s\,\rangle_k \\
\qquad\hookrightarrow_2 \langle\, \texttt{false \&\& (true \&\& true)} \curvearrowright \texttt{.}\,\rangle_k & & \qquad\hookrightarrow_2 \langle\, v_1\ \texttt{\&\&}\ e_2 \curvearrowright s\,\rangle_k \\
\qquad\quad\hookrightarrow_4 \langle\, \texttt{false} \curvearrowright \texttt{.}\,\rangle_k & & \qquad\quad\hookrightarrow_4 \langle\, v_1 \curvearrowright s\,\rangle_k
\end{array}$$

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

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 *evaluation*. 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, coming 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. Papers about $\mathbb{K}$, e.g. [19], mention very few of them and the documentation [2] is not exhaustive and complete. However, many features require the use of attributes. This section presents the list of attributes in 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 List of $\mathbb{K}$ attributes as exhaustive as possible.

About importation. What comes from the 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, 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, 𝕂 will warn the user if a symbol is declared but not used in any of the rules. The user can disable this warning for a particular symbol or cell by adding the attribute `unused`.

About symbol. Compared to an evaluation strategy defined with the attributes `strict` and `seqstrict` (Section 2.1.2.3), it is possible to develop more complex ones thanks to the attributes `result` and `hybrid`. For example, these attributes can allow lists of values to be considered as 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 𝕂 to generate fresh values and use them with fresh variables `!`Var (`freshGenerator`) and also to define binder (`binder`).

About cell. The user can choose if a cell is optional (`multiplicity="?"`) or can appear several times (`multiplicity="*"`). These attributes allow the user to design a set of cells, which type can be defined by sorts `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. Theoretically rewriting rules can be applied in any order, but 𝕂 allows the user 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 𝕂 from automatically computing the configuration in a rewriting rule. Finally, it is also possible to allow variables to be unbound in 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 𝕂 standard library

The 𝕂 standard library is composed of eight files. The file `prelude.md` is imported into any 𝕂 definition and contains only two lines that import the following two files: `domains.md` which defines the types of several usual data structures, for instance, the sorts `Bytes`, `Array`, `Map`, `Set`, `List`, `Bool`, `Int`, `String` or `Float`, and `kast.md` which corresponds to the syntax of 𝕂. Moreover, the file `rat.md` is an implementation of the rational integers, the file `substitution.md` is an implementation allowing substitution (required by the attribute `binder`), and the file `unification.md` is an implementation allowing unification. Finally, the two last files are `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` $\times$ `K` $\to$ `K` [`constructor`] and `inj` : $\forall$ $(From, To :$ `K`$)$, $From \to To$.

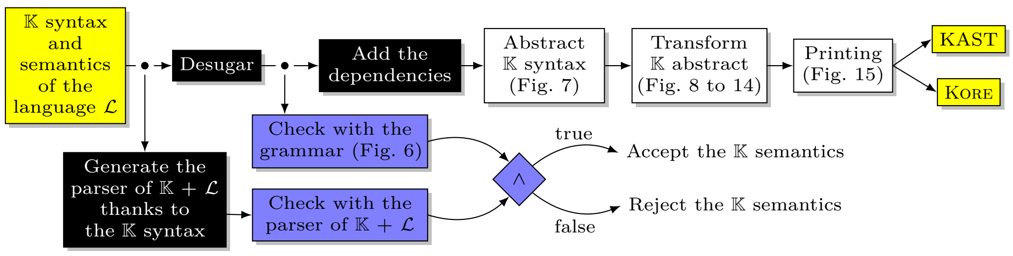
2.3 A $\mathbb{K}$ grammar

To conclude this section, we present the overview of the translation from $\mathbb{K}$ semantics which is available in Figure 5 and formalized in the next section. The $\mathbb{K}$ grammar that we consider in this article is available in 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, an 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, a configuration variable begins with `$` such as `$PGM`, a fresh variable begins with `!` and an unbound variable in a rewriting rule begins with `?`. Finally, there are three cast operators noted `:`, `::` and `:>`.

The $\mathbb{K}$ grammar in Figure 6 is almost complete. In order not to make it too heavy, we have omitted a part of the syntax allowing to declare precedence and associativity 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 declarations with at least one of the attributes related to symbolic execution ($\mathcal{A}_{\text{KProver}}$) have been deleted as well as the syntactic sugar has been simplified. For example,
`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 `...` or the fact that the rewriting arrow `=>` can be nested following Li and Gunter [15] who explain that this syntax is ambiguous syntactic sugar. We assume that the syntactic sugar is simplified 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 the syntactic sugar.



𝕂 syntax and semantics of the language ℒ
Desugar
Add the dependencies
Abstract 𝕂 syntax (Fig. 7)
Transform 𝕂 abstract (Fig. 8 to 14)
Printing (Fig. 15)
KAST
Kore
Check with the grammar (Fig. 6)
Generate the parser of 𝕂 + ℒ thanks to the 𝕂 syntax
Check with the parser of 𝕂 + ℒ
∧
true
false
Accept the 𝕂 semantics
Reject the 𝕂 semantics

Figure 5 Overview of the translation from $\mathbb{K}$ semantics.

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

⟨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` | $\boxed{\mathcal{A}_{sort}}^{+}_{,}$ `]`)?
	|	`syntax` ⟨sort⟩ `::=` ⟨sort⟩ (`[ token ]`)?
⟨terminal⟩	::=	⟨string⟩
⟨non-terminal⟩	::=	⟨sort⟩
⟨syntax-item⟩	::=	⟨terminal⟩ | ⟨non-terminal⟩
⟨separator⟩	::=	⟨string⟩
⟨syntax⟩	::=	`syntax` [`{` $\langle\text{sort}\rangle^{+}_{,}$ `}`]? ⟨sort⟩ `::=` ⟨symbol⟩`(` $\langle\text{sort}\rangle^{*}_{,}$ `)` `[` $\boxed{\mathcal{A}_{symbol}}^{*}_{,}$ `]`
	|	`syntax` [`{` $\langle\text{sort}\rangle^{+}_{,}$ `}`]? ⟨sort⟩ `::=` ⟨syntax-item⟩+ `[` $\boxed{\mathcal{A}_{symbol}}^{*}_{,}$ `]`
	|	`syntax` ⟨sort⟩ `::= r` ⟨str-of-reg-expr⟩ `[` { `token` } ∪ $\boxed{\mathcal{A}_{token}}^{*}_{,}$ `]`
	|	`syntax` ⟨sort⟩ `::= List` `{` ⟨sort⟩ `,` ⟨separator⟩ `}`
	|	`syntax` ⟨sort⟩ `::= NeList` `{` ⟨sort⟩ `,` ⟨separator⟩ `}`
⟨config-variable⟩	::=	`$`[A-Z]+
⟨initial-value⟩	::=	⟨symbol⟩ | ⟨config-variable⟩
⟨cell⟩	::=	`<` ⟨name⟩ $\boxed{\mathcal{A}_{cell}}^{*}_{,}$`>` ⟨initial-value⟩ [`:` ⟨sort⟩]? `</` ⟨name⟩`>`
	|	`<` ⟨name⟩ $\boxed{\mathcal{A}_{cell}}^{*}_{,}$`>` ⟨cell⟩+ `</` ⟨name⟩`>`
⟨configuration⟩	::=	`configuration` ⟨cell⟩+
⟨variable⟩	::=	[`?` | `!`]? [A-Z] ⟨carac⟩* | `_`
⟨pattern⟩	::=	(⟨variable⟩ | ⟨symbol⟩) [`:` ⟨sort⟩ | `::` ⟨sort⟩]?
	|	`{` ⟨pattern⟩+ `}` `:>` ⟨sort⟩
⟨rule⟩	::=	`rule` ⟨pattern⟩+ `=>` ⟨pattern⟩+ [`requires` ⟨pattern⟩+]? `[` $\boxed{\mathcal{A}_{rule}}^{*}_{,}$ `]`
⟨context⟩	::=	`context` ⟨pattern⟩+ [`requires` ⟨pattern⟩+]? (`[` $\boxed{\mathcal{A}_{context}}^{+}_{,}$ `]`)?
⟨context-alias⟩	::=	`context alias` `[` ⟨carac⟩+ `]:` ⟨pattern⟩+ (`[` $\boxed{\mathcal{A}_{context-alias}}^{+}_{,}$ `]`)?
⟨sentence⟩	::=	⟨sort-syntax⟩ | ⟨syntax⟩
	|	⟨configuration⟩ | ⟨rule⟩
	|	⟨context⟩ | ⟨context-alias⟩
⟨module⟩	::=	`module` ⟨name-module⟩ (`[` $\boxed{\mathcal{A}_{module}}^{+}_{,}$ `]`)? ⟨import⟩* ⟨sentence⟩* `endmodule`
⟨file⟩	::=	⟨require⟩* ⟨module⟩*
where $\mathcal{A}_{module}$	$\triangleq$	$\mathcal{A}_{visibility} \cup \mathcal{A}_{backend} \cup$ { `not-lr1` }
$\mathcal{A}_{symbol}$	$\triangleq$	$\mathcal{A}_{parsing} \cup \mathcal{A}_{family} \cup \mathcal{A}_{property} \cup \mathcal{A}_{strategy} \cup \mathcal{A}_{modulo} \cup \mathcal{A}_{printing} \cup \mathcal{A}_{visibility}$
$\mathcal{A}_{cell}$	$\triangleq$	$\mathcal{A}_{structure} \cup \mathcal{A}_{console} \cup$ { `unused=""`, `color=` ⟨string⟩ }
$\mathcal{A}_{context}$	$\triangleq$	{ `result` }
$\mathcal{A}_{context-alias}$	$\triangleq$	{ `result`, `context` }

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

3 Abstracting the $\mathbb{K}$ framework

This section presents a mathematical structure $\mathcal{M}$ which abstracts 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 $\mathcal{M}$ that 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) that 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 $\mathcal{A}_{\text{KProver}}$.

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 context aliases. Figure 7 shows the translation $||\ .\ ||$ of $\mathbb{K}$ syntax to the abstract structure $\mathcal{M}$. Syntactic declarations can create sorts, subtyping 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 rule can be seen as a conditional rule with the condition "true".

$$|| \texttt{syntax}\ s\ (\ \texttt{[}\ Attr\ \texttt{]}\ )?\ || = \mathcal{S}ort \leftarrow \{\ s\ \}$$
$$|| \texttt{syntax}\ s_1\ \texttt{::=}\ s_2\ (\ \texttt{[}\ Attr\ \texttt{]}\ )?\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \}\ ;\ \mathcal{R}el \leftarrow \{\ s_2 < s_1\ \}$$
$$|| \texttt{syntax}\ \texttt{\{}\ \alpha_1, ..., \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ sym\ \texttt{(}\ s_1, ..., s_x\ \texttt{)}\ \texttt{[}\ Attr\ \texttt{]}\ || \text{ with } n \geq 0 \text{ and } x \geq 0 =$$
$$\mathcal{S}ort \leftarrow \{\ \alpha\ ;\ s_1\ ;\ ...\ ;\ s_x\ \} \setminus \{\ \alpha_1, ..., \alpha_n\ \}\ ;$$
$$\mathcal{S}ym \leftarrow \{\ sym : \forall\alpha_1, ..., \forall\alpha_n,\ s_1 \times ... \times s_x \to \alpha\ [Attr]\ \}$$
$$|| \texttt{syntax}\ \texttt{\{}\ \alpha_1, ..., \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ i_1\ ...\ i_x\ \texttt{[}\ Attr\ \texttt{]}\ || \text{ with } n \geq 0 \text{ and } x \geq 1 =$$
$$|| \texttt{syntax}\ \texttt{\{}\ \alpha_1, ..., \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ t_1\text{-}...\text{-}t_n\ \texttt{(}\ s_1, ..., s_x\ \texttt{)}\ \texttt{[}\ Attr\ \texttt{]}\ ||$$
$$\text{where } t_i \in I_{terminal} \triangleq \{\ i_k \mid k \in [\![1; x]\!] \text{ and } i_k \in \langle\textbf{terminal}\rangle\ \}$$
$$s_i \in I_{non-terminal} \triangleq \{\ i_k \mid k \in [\![1; x]\!] \text{ and } i_k \in \langle\textbf{non-terminal}\rangle\ \}$$
$$|| \texttt{syntax}\ s\ \texttt{::=}\ \texttt{r}\ \langle\textbf{str-of-reg-expr}\rangle\ \texttt{[}\ Attr\ \texttt{]}\ || = \mathcal{S}ort \leftarrow \{\ s\ \}$$
$$|| \texttt{syntax}\ s_1\ \texttt{::=}\ \texttt{List}\ \texttt{\{}\ s_2\ \texttt{,}\ sep\ \texttt{\}}\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \}$$
$$|| \texttt{syntax}\ s_1\ \texttt{::=}\ \texttt{NeList}\ \texttt{\{}\ s_2\ \texttt{,}\ sep\ \texttt{\}}\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \}$$

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

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

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

Figure 7 From $\mathbb{K}$ to an abstract 7-uplet, where $X \leftarrow data$ is the set X extended with $data$.

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

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

This subsection formalizes the transformations on the previous mathematical structure $\mathcal{M}$ that correspond to the static semantics of $\mathbb{K}$. After these transformations, the 7-uplet $\mathcal{M}$ will become a quadruplet. As 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 $\mathcal{R}_{lib}$, we initially assume that $\mathcal{S}ort = \mathcal{S}ort_{lib}$, $\mathcal{R}el = \mathcal{R}el_{lib}$, $\mathcal{S}ym = \mathcal{S}ym_{lib}$ and $\mathcal{R} = \mathcal{R}_{lib}$ whereas $\mathcal{C}onfig$, $\mathcal{C}ontext$ and $\mathcal{C}ontext_{alias}$ are empty. The goal of the first transformation (Figure 8) is to check the validity of a $\mathbb{K}$ semantics.

Each symbol should be a *constructor*, a *function*, a *token*, a *bracket* or a *macro*. If this is not the case, it means that the symbol is implicitly a constructor. We therefore explicitly add the attribute `constructor`, which defines the new set $\mathcal{S}ym'$ (Figure 8). Regarding the attribute `macro`, the documentation is not precise enough. According to Li and Gunter [14], 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 such as the attributes `strict` and `seqstrict` are incompatible with `function`, while a `bracket` symbol of a given sort has only one argument of that sort.

$$|| \, (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias}) \, ||_{\texttt{check-input}} =$$
$$(\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym', \mathcal{C}onfig', \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias})$$

where

$$\begin{array}{lcl}
\mathcal{C} & \triangleq & \{ \; s \in \mathcal{S}ym \mid \texttt{constructor} \in Attr(s) \; \} \\
\mathcal{F} & \triangleq & \{ \; s \in \mathcal{S}ym \mid \texttt{function} \in Attr(s) \; \} \\
\mathcal{T} & \triangleq & \{ \; s \in \mathcal{S}ym \mid \texttt{token} \in Attr(s) \; \} \\
\mathcal{B} & \triangleq & \{ \; s \in \mathcal{S}ym \mid \texttt{bracket} \in Attr(s) \; \} \\
\mathcal{M}a & \triangleq & \{ \; s \in \mathcal{S}ym \mid \texttt{macro} \in Attr(s) \; \} \\
\mathcal{W} & \triangleq & \mathcal{S}ym \setminus (\mathcal{C} \cup \mathcal{F} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{M}a) \\
\mathcal{W}' & \triangleq & \{ \; n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha \; [Attr \cup \{\texttt{constructor}\}] \mid \; n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha \; [Attr] \in \mathcal{W} \; \} \\
\mathcal{S}ym' & \triangleq & \mathcal{C} \cup \mathcal{F} \cup \mathcal{T} \cup \mathcal{B} \cup \mathcal{M}a \cup \mathcal{W}' \\
\mathcal{C}onfig' & \triangleq & [k]_{\mathtt{K}}(\texttt{\$PGM}) \text{ if } \mathcal{C}onfig = \emptyset; \; \mathcal{C}onfig \text{ if } \mathrm{card}(\mathcal{C}onfig) = 1
\end{array}$$

Constraints:

The set $\{ \; \mathcal{C}, \; \mathcal{F}, \; \mathcal{T}, \; \mathcal{B}, \; \mathcal{M}a, \; \mathcal{W} \; \}$ must be a partition of $\mathcal{S}ym$.
$\mathcal{C}onfig$ is the empty set or a singleton, where each cell have a unique name.
For all $n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha \; [Attr] \in \mathcal{F}$, $\{ \; \texttt{strict}, \texttt{seqstrict} \; \} \cap Attr = \emptyset$.
For all $n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha \; [Attr] \in \mathcal{B}$, $\overrightarrow{v} = \overrightarrow{0}$ and $\overrightarrow{t} = \alpha$.

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

Generate evaluation strategy rules thanks to contexts and context aliases. We have seen that to define evaluation strategies in $\mathbb{K}$, we can define context aliases, 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 formalizing these two functions.

Generate evaluation strategy rules thanks to attributes. We formalize the rule generation from the attributes `strict` and `seqstrict` (when the attributes `result` and `hybrid` are not also used) in Figure 9, where $\texttt{nbArg}(sym)$ is the number of argument(s) of the symbol sym and n is implicitly used to denote the arity of a symbol. For example, rules 1. and 2. (Figure 3) illustrate the translation formalized in Figure 9.

$$
\begin{array}{l}
||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{C}onfig,\ \mathcal{R})\ ||_{\texttt{generate-strategy}} = (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym',\ \mathcal{C}onfig,\ \mathcal{R}') \\
\quad \text{where} \\
\qquad \begin{array}{lcl}
\mathcal{S}_{\texttt{strict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{strict} \in Attr(s)\ \} \\
\mathcal{S}_{\texttt{seqstrict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{seqstrict} \in Attr(s)\ \} \\
\mathcal{F}reezer & \triangleq & \{\ \circledast^{n_j}_{sym} :\ \underbrace{\texttt{K} \times \ldots \times \texttt{K}}_{\texttt{nbArg}(sym)-1} \to \texttt{KItem}\ [\texttt{constructor}\ ] \\
 & & \quad \mid\ \text{and } \texttt{strict}\ (n_1,\ldots,n_k) \in Attr(sym) \\
 & & \quad\ \ \text{or } \texttt{seqstrict}\ (n_1,\ldots,n_k) \in Attr(sym)\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{S}ym\ \cup\ \mathcal{F}reezer \\
\mathcal{R}' & \triangleq & \mathcal{R}\ \cup\ \mathcal{R}_{\texttt{strict}}\ \cup\ \mathcal{R}_{\texttt{seqstrict}}
\end{array} \\
\quad \textbf{Constraint:} \\
\qquad \text{Every argument of the attribute } \texttt{strict} \text{ or } \texttt{seqstrict} \text{ should be in } [1; arity(sym)].
\end{array}
$$

$$
\begin{array}{l}
\mathcal{R}_{\texttt{strict}} \text{ is composed of the following rewriting rules:} \\
\quad \text{For all } sym \in \mathcal{S}_{\texttt{strict}} \text{ such that } \texttt{strict}\ (n_1,\ldots,n_j) \in Attr(sym), \text{ for all } k \in \{\ n_1,\ \ldots,\ n_j\ \}: \\
\quad sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\circledast^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ], \text{ where } c \triangleq \neg\ (\texttt{isKResult}\ E_k) \\
\quad E_k \curvearrowright (\circledast^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ], \text{ where } c \triangleq \texttt{isKResult}\ E_k \\
\hline
\mathcal{R}_{\texttt{seqstrict}} \text{ is composed of the following rewriting rules:} \\
\quad \text{For all } sym \in \mathcal{S}_{\texttt{seqstrict}} \text{ such that } \texttt{seqstrict}\ (n_1,\ldots,n_j) \in Attr(sym), \text{ for all } k \in \{\ n_1,\ \ldots,\ n_j\ \}: \\
\quad sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\circledast^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ] \\
\qquad \text{where } c \triangleq \texttt{isKResult}\ E_1 \wedge \ldots \wedge \texttt{isKResult}\ E_{k-1} \wedge \neg\ (\texttt{isKResult}\ E_k) \\
\quad E_k \curvearrowright (\circledast^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ], \text{ where } c \triangleq \texttt{isKResult}\ E_k
\end{array}
$$

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

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

Generate implicit cells. Now that we have generated the full configuration, we formalize the completion of the rewriting rules in Figure 10. For instance, the result of 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, rule 4. from Figure 3 becomes:
$\texttt{rule}\ \langle\ \langle\ \langle\ \texttt{false \&\& \_} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}\ \texttt{=>}\ \langle\ \langle\ \langle\ \texttt{false} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$.

$$
\begin{array}{l}
||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{C}onfig,\ \mathcal{R})\ ||_{\texttt{add-cell}} = (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{C}onfig,\ \mathcal{R}') \\
\quad \text{where} \\
\begin{array}{lll}
 & \mathcal{R}'\ \triangleq & \{\ ||\ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ and } x \triangleq \text{mgconf } \mathcal{C}onfig\ \} \\
\text{with} & \multicolumn{2}{l}{\text{mgconf } (<C>(Cell_1,\ \ldots,\ Cell_n)) \triangleq <C>(\text{mgconf } Cell_1,\ \ldots,\ \text{mgconf } Cell_n)} \\
 & \multicolumn{2}{l}{\text{mgconf } ([C]_s(v)) \triangleq [C]_s(f), \text{ where } f \text{ is a fresh variable}} \\
\text{with} & \multicolumn{2}{l}{||\ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x\ \triangleq l \overset{c}{\hookrightarrow} r\ [Attr], \text{ if the head symbol of } l \text{ is a function symbol}} \\
 & \multicolumn{2}{l}{\qquad\qquad\qquad\quad\ \ l \overset{c}{\hookrightarrow} r\ [Attr], \text{ if } \texttt{anywhere} \in Attr} \\
 & \multicolumn{2}{l}{\qquad\qquad\qquad\quad\ \ \overline{l}^{\,x} \overset{c}{\hookrightarrow} \overline{r}^{\,x}\ [Attr], \text{ otherwise}} \\
\text{with} & \multicolumn{2}{l}{\overline{p}^{\,x} \triangleq x\ [\ \overrightarrow{[C]_s(y)} \setminus \overrightarrow{[C]_s(v)}\ ] \text{ where } \overrightarrow{[C]_s(v)} \text{ are the leaves appearing in } p}
\end{array}
\end{array}
$$

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

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 in Figure 11, where GT $\triangleq$ `GeneratedTop` and GC $\triangleq$ `GeneratedCounter`.

$$
\begin{array}{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}') \\
\quad \text{where} \\
\begin{array}{lll}
\qquad \mathcal{S}ort' & \triangleq & \mathcal{S}ort \cup \{\ \texttt{Sort}C \mid \texttt{<}C\texttt{>}(c_1, \ldots, c_n) \in \mathcal{C}onfig\ \} \\
\qquad \mathcal{S}ym_{cell} & \triangleq & \{\ C : \text{Type}\ (\ \texttt{<}C\texttt{>}(c_1, \ldots, c_n)\ )\ [\texttt{constructor}\ ] \mid \texttt{<}C\texttt{>}(c_1, \ldots, c_n) \in \mathcal{C}onfig\} \\
\qquad \mathcal{S}ym_{init} & \triangleq & \{\ \texttt{init}C : \texttt{Sort}C\ [\texttt{function}, \texttt{initializer}\ ] \mid \texttt{<}C\texttt{>}(c_1, \ldots, c_n) \in \mathcal{C}onfig\} \\
\qquad \mathcal{S}ym_{get} & \triangleq & \{\ \texttt{getGC} : \texttt{SortGT} \to \texttt{SortGC}\ [\texttt{function}\ ]\ \} \\
\qquad \mathcal{S}ym' & \triangleq & \mathcal{S}ym\ \cup\ \mathcal{S}ym_{cell}\ \cup\ \mathcal{S}ym_{init}\ \cup\ \mathcal{S}ym_{get} \\
\qquad \mathcal{R}_{init} & \triangleq & \{\ \texttt{init}C \hookrightarrow C((\text{GetInit}\ c_1), \ldots, (\text{GetInit}\ c_n))\ [\texttt{initializer}\ ] \\
 & & \quad \mid \texttt{<}C\texttt{>}(c_1, \ldots, c_n) \in \mathcal{C}onfig\} \\
\qquad \mathcal{R}' & \triangleq & \mathcal{R}\ \cup\ \mathcal{R}_{init} \cup\ \{\ \texttt{getGC}(\text{GT}(\text{X,V})) \hookrightarrow \text{V}\ \}
\end{array} \\
\quad \text{with} \quad \text{Type}\ (\ \texttt{<}X\texttt{>}(c_1, \ldots, c_n)\ ) \triangleq \text{RetType}(c_1) \times \ldots \times \text{RetType}(c_n) \to \texttt{Sort}X \\
\quad \text{with} \quad \text{RetType}\ (\ \texttt{<}X\texttt{>}(c_1, \ldots, c_n)\ ) \triangleq \texttt{Sort}X \\
\qquad\qquad\ \ \text{RetType}\ (\ [X]_s(v)\ ) \triangleq s \\
\quad \text{with} \quad \text{GetInit}\ (\ \texttt{<}X\texttt{>}(c_1, \ldots, c_n)\ ) \triangleq \texttt{init}X \\
\qquad\qquad\ \ \text{GetInit}\ (\ [X]_s(v)\ ) \triangleq v
\end{array}
$$

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

For example, the symbol `initK : SortK` and the rule `initK` $\hookrightarrow$ `K $PGM` are generated as well as the symbols GT : `SortT` $\times$ `SortGC` $\to$ `SortGT` and `T` : `SortK` $\to$ `SortT`.

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

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

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

Manage the fresh values. When using a $\mathbb{K}$ rewriting rule, any occurrence of a fresh variable `!`X is replaced by the current value presents in the cell `GeneratedCounter` and the current value of the cell `GeneratedCounter` is replaced by a new one. For instance, the rewriting rule `rule` $\langle\ \langle\ \langle\ \texttt{foo X} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle\ !\text{X} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ 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 type-related symbols and extend the typing hierarchy. Implicitly, every user-defined sort is a sub-sort of `KItem`. Moreover, $\mathbb{K}$ generates projection symbols and predicate symbols such as `isKResult`, as shown in Figure 13.

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

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 injections (`inj`) to get full well-typed terms. This step takes into account 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 sort `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. For example, 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
$\{\ n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{S}ym \mid$ `constructor` $\in Attr$ and $\alpha \in \mathcal{S}ort_{lib}\ \}$ should be empty. That is the reason why we named a sort **MyBool** and not **Bool** in Figure 2.

Checking the constraints on rewriting rules. Finally, each rewriting rule needs to respect the BNF in Figure 14 and the sort of every condition must be boolean.

a	$::=$	$x \mid (\sigma\ a \ldots a)$	x is a variable
b	$::=$	$x \mid (\sigma\ b \ldots b) \mid (f\ b \ldots b)$	σ is a `constructor` symbol
c	$::=$	$f\ b \ldots b$	f is a `function` symbol
$rule$	$::=$	$\sigma\ a \ldots a \hookrightarrow b \mid \sigma\ a \ldots a \overset{c}{\hookrightarrow} b$	
		$\mid f\ a \ldots a \hookrightarrow b \mid f\ a \ldots a \overset{c}{\hookrightarrow} b$	

Figure 14 Constraints on rewriting rules.

We have 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 $\mathbb{K}$ developpers. It seems possible to print a (almost always valid) new $\mathbb{K}$ file after each transformation but this has not been implemented. So far only the following attributes have not been taken into account during 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 corresponding to the axioms of injectivity, non-overlapping and exhaustivity of constructors.

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

This section presents the $\lambda\Pi$-calculus modulo theory and Dedukti, a logical framework based on it. Then, we formalize the translation from $\mathbb{K}$ to the $\lambda\Pi$-calculus modulo

$\Vert\, (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\, \Vert_{\text{KORE}} =$	
`sort` $s\{\alpha_1,\ldots,\alpha_n\}$ [ ]	for all $s \in \mathcal{S}ort \setminus \mathcal{S}ort_{lib}$
`hooked-sort` $s\{\alpha_1,\ldots,\alpha_n\}$ [ ]	for all $s \in \mathcal{S}ort_{lib}$
`symbol` $sym\{\alpha_1,\ldots,\alpha_n\}(\theta_1,\ldots,\theta_m)$: θ' [Attr]	for all $sym \in \mathcal{S}ym \setminus \mathcal{S}ym_{lib}$
`hooked-symbol` $sym\{\alpha_1,\ldots,\alpha_n\}(\theta_1,\ldots,\theta_m)$: θ' [Attr]	for all $sym \in \mathcal{S}ym_{lib}$
`axiom` {R} `\exists` {R}(x_2 : θ_2, `\equals` {θ_2, R}(x_2 : θ_2, `inj` {θ_1, θ_2}(x_1 : θ_1))) [`subsort` (θ_1, θ_2)]	for all $\theta_1 < \theta_2 \in \mathcal{R}el$
`axiom` {R} `\exists` {R}(x : θ, `\equals` {θ, R}(x : θ, $sym\ x_1\ \ldots\ x_n$)) [`total`]	for all $sym \in \mathcal{S}_{\texttt{total}}$
`axiom` {$\alpha_1,\ldots,\alpha_n$} φ_{sym} [`constructor`]	for all $sym \in \mathcal{S}_{\texttt{constructor}}$
`axiom` {R} `\equals` {θ, R}($sym(sym(x_1 : \theta,\ x_2 : \theta),\ x_3 : \theta)$, $sym(x_1 : \theta,\ sym(x_2 : \theta,\ x_3 : \theta))$) [`assoc`]	for all $sym \in \mathcal{S}_{\texttt{assoc}}$
`axiom` {R} `\equals` {θ, R}($sym(x_1 : \theta,\ x_2 : \theta)$, $sym(x_2 : \theta,\ x_1 : \theta)$) [`comm`]	for all $sym \in \mathcal{S}_{\texttt{comm}}$
`axiom` {R} `\equals` {θ, R}($sym(e,\ x : \theta),\ x : \theta$) [`unit`]	
`axiom` {R} `\equals` {θ, R}($sym(x : \theta,\ e),\ x : \theta$) [`unit`]	for all $sym \in \mathcal{S}_{\texttt{unit}}$
`axiom` {R} `\equals` {θ, R}($sym(x : \theta,\ x : \theta),\ x : \theta$) [`idem`]	for all $sym \in \mathcal{S}_{\texttt{idem}}$
`axiom` {R} `\implies` {R}(c, `\equals` {R, R}(l, r)) [$Attr$]	for all $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{function}}$
`axiom` {R} `\rewrites` {R}(`\and` {R}(c, l), r) [$Attr$]	for all $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{constructor}}$

where $\mathcal{S}_a \triangleq \{\ s \in \mathcal{S}ym \mid a \in Attr(s)\ \}$ and $\mathcal{R}_a \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \mid a \in Attr(head(l))\ \}$

Figure 15 The printer to KORE.

THEORY. This translation has been implemented in a tool written in OCAML, named KAMELO, which is presented in the next section.

4.1 The $\lambda\Pi$-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 [10]. $\lambda\Pi{\equiv}_{\mathcal{T}}$ is an extension of the λ-calculus with dependent types and a primitive notion of computation defined thanks to rewriting rules [11]. The syntax as well as the typing rules that define the $\lambda\Pi{\equiv}_{\mathcal{T}}$ are available in Figure 16, where the typing judgment $\Gamma \vdash t : A$ means that the term t has type A with respect to the context Γ. The specific typing judgment $\Gamma \vdash A : \textbf{Type}$ 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, for any rewriting rule $l \hookrightarrow r \in \mathcal{R}$, $FV(r) \subseteq FV(l)$ holds (where $FV(p)$ is the set of free variables of p). The use of such a rewriting rule requires that for any substitution σ, the instantiation of its left-hand side $l\sigma$ and the instantiation of its right-hand side $r\sigma$ 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	:=	$\textbf{Type} \mid \textbf{Kind}$	*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

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

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

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

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

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

where $s \in \{\textbf{Type}\ ; \textbf{Kind}\}$, $B\{x\backslash a\}$ 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 β-reduction but also on the rewriting system $\mathcal{R}$. Moreover, in order to have the decidability of the type-checking, the condition $A \equiv_{\beta\mathcal{R}} B$ of 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. Indeed, thanks to the rule (var), only well-formed elements in the context can be used when doing a proof. This presentation has been proved equivalent to the usual presentations by Dowek [12].

4.2 Dedukti

Dedukti [4, 5] 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 Dedukti, facilitating the interoperability of proofs between various formal tools [9, 22]. In this section, we only present the features available in Dedukti needed in this article.

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

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 not be reduced by any rewriting rule.

Rewriting rules. A Dedukti rule is written `rule` *LHS* $\hookrightarrow$ *RHS* in which the free variables are noted `$x`, `$y`, etc. As in $\mathbb{K}$, 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 abstract $\mathbb{K}$ to the $\lambda\Pi$-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 the $\lambda\Pi$-calculus modulo theory.

$$
\begin{array}{ll}
\multicolumn{2}{l}{||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\ ||_{\lambda\Pi\equiv_{\mathcal{T}}} =} \\
\quad \mathtt{K} : \mathbf{Type} & \\
\quad s : \mathtt{K} & \text{for all } s \in \mathcal{S}ort \setminus \{\mathtt{K}\} \\
\quad n : \Pi(a_1 : \alpha_1), ..., \Pi(a_n : \alpha_n), \Pi(t_1 : \tau_1), ..., \Pi(t_n : \tau_n), \alpha & \text{for all } n : \forall \overrightarrow{\alpha}, \overrightarrow{\tau} \to \alpha\ [Attr] \in \mathcal{S}ym \\
\quad l \hookrightarrow r & \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional} \\
\quad ||\ \mathcal{R}_{conditional}\ ||_{\mathtt{CTRS}} & \\
\quad ||\ (\mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R}_{strategy})\ ||_{\mathtt{strategy}} & \\
\\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{unconditional} \triangleq \{\ l \hookrightarrow r \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } c = \mathtt{true}\ \}} \\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{conditional} \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } c \neq \mathtt{true} \text{ and } \{\ \mathtt{heat},\ \mathtt{cool}\ \} \cap Attr = \emptyset\ \}} \\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{strategy} \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } \mathtt{heat} \in Attr \text{ or } \mathtt{cool} \in Attr\ \}}
\end{array}
$$

Figure 17 From abstract $\mathbb{K}$ to the $\lambda\Pi{\equiv}_{\mathcal{T}}$.

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 $\{\ l \hookrightarrow r \mid \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional}\ \} \cup ||\ \mathcal{R}_{conditional}\ ||_{\mathtt{CTRS}} \cup ||\ (\mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R}_{strategy})\ ||_{\mathtt{strategy}}$. The translation function $||\ .\ ||_{\mathtt{CTRS}}$ is explained in Section 4.3.1 and the translation function $||\ .\ ||_{\mathtt{strategy}}$ is explained in 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 conditional rewriting rules are not primitive 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 illustrate 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 in 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 a computation (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. In Figure 18, rule (0) rewrites a term whose head symbol is max into a term using the corresponding extended version of arity 4, $\flat max$ here, where all boolean arguments are $\flat$, indicating 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 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: implement an algorithm that determines the complementary condition or consider that all conditions necessarily reduce to either `true` or `false`. According to the expressiveness of the conditions that can be written in $\mathbb{K}$ (Figure 14), we add the following hypothesis: any boolean function 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 in Figure 19. This case is formalized in 5.(c) (Figure 20).

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*`)`, with which we do not yet take into account.

(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.

4.3.1.2 From a CTRS to a TRS: Formalization

We present the translation, noted $||\ .\ ||_{\texttt{CTRS}}$ previously, as an algorithm in 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 *head symbol of a rule*. We note $head_{\texttt{<k>}}$ the function computing the head symbol of the cell `<k>`, for a given rule, without considering the symbols `.`, $\curvearrowright$ and `inj`. Now, we are able to define the function returning the head symbol of a rule:

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

We also define the set of rules noted $\mathcal{C}_\sigma$, in which the rules share the same head symbol σ, that is $\mathcal{C}_\sigma \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid head\ (l \overset{c}{\hookrightarrow} r\ [Attr]) = \sigma\ \}$. We assume that, if there is a rule with the attribute `owise` in $\mathcal{C}_\sigma$ for a given σ, all the other rules in $\mathcal{C}_\sigma$ must have a condition.

After calculating each set $\mathcal{C}_\sigma$ from $\mathfrak{R}$, we run the algorithm presented in 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 $\sigma$, noted $mglhs_\sigma$.
3. Generate the extended symbol $\flat\sigma$ of type $T_1 \to \dots \to T_{n-1} \to \flat$Bool $\to \dots \to \flat$Bool $\to T_n$,
   with $X$ argument(s) of type $\flat$Bool, where $\flat$Bool $=$ Bool $\cup \{\flat\}$, and $\sigma$ of type $T_1 \times \dots \times T_{n-1} \to T_n$.
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 $c \neq$ true and 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 $c =$ true and owise $\notin Attr$:
      - Generate the reduction rule: $l\ [\ update_{same}(\sigma, \flat\sigma, \_)\ ]_\sigma \hookrightarrow r$
   c. If $c =$ true and 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 head symbol $\sigma$ 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 $init\text{-}config$) $[\ [\texttt{k}]_{\texttt{K}}(x) \setminus [\texttt{k}]_{\texttt{K}}((\texttt{inj}^{\texttt{KItem}}_{RetType(\sigma)}\ \sigma\ f_1\ \dots\ f_{arity(\sigma)}) \curvearrowright L)\ ]$,
where $init\text{-}config$ is the initial configuration, and $f_i$ and $L$ are fresh variables.
* $update_{diff}(\sigma, \flat\sigma, s_1, i, s_2) = \flat\sigma\ x_1\ \dots\ x_{arity(\sigma)+X}\ with\ x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \le j \le arity(\sigma) \\ s_1 & \text{if } j = arity(\sigma) + i \\ s_2 & \text{otherwise} \end{cases}$
* $update_{same}(\sigma, \flat\sigma, s) = \flat\sigma\ x_1\ \dots\ x_{arity(\sigma)+X}\ with\ x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \le j \le arity(\sigma) \\ s & \text{otherwise} \end{cases}$
```

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 stop the computation. 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 stuck execution[1] (on the left).

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

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

Intuitively, these rules are used to ensure that E_1 is of a specific sort in order to allow or not its evaluation. The idea of our new encoding is to specialize some terms of the rules, i.e. to refine the pattern-matching in order to ensure the desired type. For example, rule 2. in Figure 3 becomes $\langle\ (\texttt{inj}^{\texttt{KItem}}_{\texttt{Bool}}\ E_1) \curvearrowright (\text{❄}^1_{\texttt{\&\&}}\ E_2) \curvearrowright S\ \rangle_k \hookrightarrow \langle\ (\texttt{inj}^{BExp}_{\texttt{Bool}}\ E_1)\ \texttt{\&\&}\ E_2 \curvearrowright S\ \rangle_k$, where we force E_1 to have the sort `Bool`. Thus this rule can be used only if the term E_1 is a fully computed Boolean expression. Morevoer, rule 1. in Figure 3 becomes the single rule $\langle\ (X_1\ \texttt{\&\&}\ X_2)\ \texttt{\&\&}\ E_2 \curvearrowright S\ \rangle_k \hookrightarrow \langle\ (X_1\ \texttt{\&\&}\ X_2) \curvearrowright (\text{❄}^1_{\texttt{\&\&}}\ E_2) \curvearrowright S\ \rangle_k$ 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 symbols with the attribute `bracket`, because these ones disappear during the compilation process of $\mathbb{K}$. The full formalization is available in Figure 22.

$$
\begin{array}{l}
||\ (\mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\ ||_{\texttt{strategy}} = \mathcal{R}' \\
\quad \text{where} \\
\begin{array}{rcl}
\mathcal{S}ub & \triangleq & \{\ s \mid s < \texttt{KResult} \in \mathcal{R}el\ \} \\
\mathcal{R}_{\texttt{cool}} & \triangleq & \{\ r \in \mathcal{R} \mid \texttt{cool} \in Attr(r)\ \} \\
\mathcal{R}'_{\texttt{cool}} & \triangleq & \{\ (l \hookrightarrow r)\ [\ x \setminus \texttt{inj}^{\texttt{KItem}}_{s}\ x\ ] \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\texttt{cool}} \text{ where } c \triangleq (\texttt{isKResult}\ x),\ s \in \mathcal{S}ub\ \} \\
\mathcal{R}_{\texttt{heat}} & \triangleq & \{\ r \in \mathcal{R} \mid \texttt{heat} \in Attr(r)\ \} \\
\mathcal{S}^{s_2}_{s_1} & \triangleq & \{\ \texttt{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)\ \} \\
\mathcal{P}^{s_2}_{s_1} & \triangleq & \{\ \texttt{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 \text{if } \alpha = s_1 \text{ and } \{\ \texttt{constructor},\ \texttt{token}\ \} \cap Attr \neq \emptyset\ \} \\
\mathcal{R}'_{\texttt{heat}} & \triangleq & \{\ (l \hookrightarrow r)\ [\ x_1,\ \dots\ ,\ x_k \setminus \texttt{inj}^{\texttt{KItem}}_{s_1}\ x_1,\ \dots\ ,\ \texttt{inj}^{\texttt{KItem}}_{s_k}\ x_k\ ]\ [\ \texttt{inj}^{s_2}_{s_1}\ x \setminus t\ ] \\
 & & \qquad \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\texttt{heat}} \text{ and } (s_1, \dots, 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\quad \text{where } c \triangleq (\texttt{isKResult}\ x_1 \wedge \dots \wedge \texttt{isKResult}\ x_k \wedge \neg\ (\texttt{isKResult}\ \texttt{inj}^{s_2}_{s_1}\ x)\ \} \\
\mathcal{R}' & \triangleq & \mathcal{R} \setminus (\mathcal{R}_{\texttt{heat}}\ \cup\ \mathcal{R}_{\texttt{cool}}) \cup (\mathcal{R}'_{\texttt{heat}} \cup \mathcal{R}'_{\texttt{cool}})
\end{array}
\end{array}
$$

Figure 22 Specialization of the evaluation strategy rules.

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

4.3.3 Semantics preservation

The soundness 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 the 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 evaluation rules, only one rule has the attribute `owise` and in this case, other rules must have a condition.

The two following parts present the soundness and completeness statements, thanks to the function $|\ .\ |$ which is translated a $\mathbb{K}$ term into a DEDUKTI one by induction on Term($\mathbb{K}$):

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

4.3.3.1 Soundness

The next conjecture helps to assert that any derivation in $\mathbb{K}$ is also a derivation in DEDUKTI.

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

▶ **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 $\mathcal{Flat}$ the set of every term starting with $\flat$.
We note $\mathcal{Ghost}$ the set of every term having at least one symbol in $\mathcal{Flat}$.

▶ **Lemma 1.** *$|\ .\ | : Term(\mathbb{K}) \to Term(\text{DEDUKTI}) \setminus \mathcal{Ghost}$ is a bijection.*

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

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

The following conjecture states that any derivation in DEDUKTI is also a derivation in $\mathbb{K}$, except if the derivation begins or ends with a term generated by the Viry encoding.

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

▶ **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.*

5 Implementation and examples

This section focuses on the implementation of the translations presented in the previous section, i.e. on the tool KAMELO [1] 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 $\mathbb{K}$ team. Like the command `krun`, also implemented by the $\mathbb{K}$ team, KAMELO allows programs translated into KORE to be executed in DEDUKTI thanks to the $\mathbb{K}$ semantics translated into KORE. KAMELO implements the translation formalized in Figure 20 and Figure 22.

Moreover, it is the responsibility of each backend to implement the $\mathbb{K}$ 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 $\mathbb{K}$ 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 a simple while-language similar to IMP in [25], 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 computing the GCD 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 GCD.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 [1] is joined by some tests as the one presented here.

6 Conclusion

This article presents a paper formalization of the translation from $\mathbb{K}$ 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 [8, 9], 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 $\mathbb{K}$ 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 GitLab of KaMeLo. URL: `https://gitlab.com/semantiko/kamelo`.

2 Website of $\mathbb{K}$. URL: `https://kframework.org/`.

3 Website of Sail. URL: `https://www.cl.cam.ac.uk/~pes20/sail/`.

4 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 λΠ-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`.

5 Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.FSCD.2021.20`.

6 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:10.1145/2676726.2676982`.

7 Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR: The System. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Massachusetts, USA, November 28-30, 1988*, pages 14–24. ACM, 1988. `doi:10.1145/64135.65005`.

8 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. `doi:10.1007/978-3-319-46750-4_26`.

9 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*, page 532, Brasília, Brazil, September 2017. `doi:10.1007/978-3-319-66107-0_9`.

10 D. Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.

11 Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320, 1990.

12 Gilles Dowek. Interacting Safely with an Unsafe Environment. *CoRR*, abs/2107.07662, 2021. `arXiv:2107.07662`, `doi:10.4204/EPTCS.337.3`.

13 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:10.1145/2813885.2737979`.

14 Liyi Li and Elsa L. Gunter. IsaK: A Complete Semantics of $\mathbb{K}$. Technical report, University of Illinois at Urbana-Champaign, June 2018. URL: `https://hdl.handle.net/2142/100116`.

15 Liyi Li and Elsa L. Gunter. IsaK-static: A complete static semantics of $\mathbb{K}$. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings*, pages 196–215. Springer-Verlag Berlin Heidelberg, 2018. `doi:10.1007/978-3-030-02146-7_10`.

16 Liyi Li and Elsa L. Gunter. A Complete Semantics of 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.

17 Dominic Mulligan, Scott Owens, Kathryn Gray, Tom Ridge, and Peter Sewell. Lem: Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, 49, August 2014. doi:10.1145/2628136.2628143.

18 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:10.1145/2737924.2737991.

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

20 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.

21 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. doi:10.1145/2983990.2984027.

22 François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. *Electronic Proceedings in Theoretical Computer Science*, 274:57–71, July 2018. doi:10.4204/eptcs.274.5.

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. doi:10.1006/jsco.1999.0288.

25 Glynn Winskel. *The formal semantics of programming languages - an introduction.* Foundation of computing series. MIT Press, 1993.

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