Vers une traduction de K en Dedukti

Amélie Ledein, Valentin Blot, Catherine Dubois

Vers une traduction de $\mathbb{K}$ en Dedukti

Amélie Ledein[1]*, Valentin Blot[1], Catherine Dubois[2]

[1] Laboratoire Méthodes Formelles, Inria, Université Paris-Saclay
[2] Samovar, ENSIIE

$\mathbb{K}$ est un *framework* sémantique permettant de décrire formellement des sémantiques de langages de programmation. C'est aussi un environnement qui offre différents outils pour aider à la programmation avec les langages spécifiés dans le formalisme. Il est par exemple possible d'exécuter des programmes ou encore de vérifier certaines propriétés sur ceux-ci à l'aide de l'outil KProver. $\mathbb{K}$ repose sur une logique du 1er ordre munie d'une application entre formules et d'opérateurs de point fixe, d'égalité, de typage ainsi que d'un opérateur similaire à l'opérateur "next" des logiques temporelles. Ce dernier opérateur permet d'encoder la sémantique des programmes par la réécriture.

Dedukti est un *framework* logique permettant l'interopérabilité des preuves entre différents outils de preuve formelle. Il possède des plugins d'import et d'export pour des systèmes de preuve aussi divers que Coq, PVS ou encore Isabelle/HOL. Il repose sur le $\lambda\Pi$-calcul modulo théorie, une extension de la théorie des types par l'ajout de règles de réécriture dans la relation de conversion. La flexibilité de ce *framework* logique permet d'encoder de nombreuses théories comme la logique du 1er ordre ou la théorie des types simples.

Dans cet article, nous présentons KaMeLo, un outil de traduction de $\mathbb{K}$ vers Dedukti. Cet outil a pour objectif, à plus long terme, de permettre la vérification des preuves faites au sein de $\mathbb{K}$, et la réutilisation de sémantiques formelles de langages de programmation ainsi que des propriétés sur leurs programmes au sein de nombreux systèmes de preuve, via Dedukti.

1 Introduction

Les méthodes formelles ont pour principal objectif l'obtention d'une plus grande confiance dans les programmes informatiques. Mais avant même de pouvoir vérifier un programme, celui-ci doit être écrit dans un langage de programmation dont il est indispensable de connaître précisément la syntaxe et la sémantique, et donc de disposer, dans un premier temps, d'une formalisation de la sémantique du langage de programmation utilisé pour écrire le programme que nous souhaitons vérifier. De nombreux outils permettent d'écrire des sémantiques formelles, comme par exemple Centaur [12], ASF+SDF [29], Ott [27], Sail [7], Lem [23] ou encore $\mathbb{K}$ [6]. Dans cet article, nous nous intéressons uniquement à ce dernier, puisque actuellement, il existe un grand nombre de sémantiques de langage de programmation écrites en $\mathbb{K}$, comme celles de Java [11], C [20] ou encore JavaScript [24]. Une fois la sémantique d'un langage spécifiée, $\mathbb{K}$ offre la possibilité d'exécuter un programme écrit dans ce langage, mais également la possibilité de vérifier certaines propriétés - exprimées sous la forme de propriétés d'atteignabilité - sur ce programme, à l'aide du prouveur automatique KProver [28].

Nous nous intéressons à la traduction de sémantiques écrites en $\mathbb{K}$ vers Dedukti, un *framework* logique permettant l'interopérabilité des preuves entre différents outils de preuve formelle. Cela permettrait d'exécuter au sein de Dedukti un programme écrit avec le langage formalisé, de vérifier les preuves établies par le KProver, voire faire cette preuve avec Dedukti, si le KProver a échoué, et aussi de vérifier formellement des propriétés sur le langage formalisé avec $\mathbb{K}$.

*L'auteure est financée par Digicosme.

Cet article s'intéresse à la traduction d'une sémantique écrite en 𝕂 vers Dedukti, afin de pouvoir exécuter dans Dedukti des programmes écrits dans le langage décrit par cette sémantique. 𝕂 offre de nombreuses facilités pour écrire une sémantique, comme par exemple des attributs pour spécifier une stratégie d'évaluation. La sémantique écrite par l'utilisateur est traduite dans le langage intermédiaire nommé Kore, qui sera notre format d'entrée pour la traduction vers Dedukti. Nos contributions, présentées dans cet article résident, tout d'abord, dans l'étude systématique du langage intermédiaire Kore. En effet, aucun article n'a été encore publié sur ce sujet, et très peu de documentation existe à ce jour. Nous avons donc échangé avec l'équipe 𝕂 pour conforter notre compréhension de Kore. De plus, nous formalisons la transformation effectuée par KaMeLo, un outil en cours de développement permettant de traduire une sémantique écrite en 𝕂 dans Dedukti. La correction de la traduction n'est pas démontrée dans cet article. Informellement, notre traduction cherche à assurer que le programme exécuté dans le framework 𝕂 et le programme exécuté dans Dedukti ont le même comportement - par exemple, calculent la même valeur ou donnent le même état final - si le langage décrit est déterministe.

La vérification des objets de preuve générés par le KProver, ainsi que l'encodage des fondements théoriques de 𝕂 dans ceux de Dedukti ne sont pas traités dans cet article et feront l'objet de travaux futurs. La traduction présentée ici est néanmoins nécessaire pour exécuter un programme et sera réutilisée pour la vérification des preuves.

Après une brève présentation du λΠ-calcul modulo théorie (Section 2) et de son implémentation Dedukti (Section 3), nous présentons la logique sous-jacente de 𝕂, la Matching Logic (Section 4), ainsi que le framework 𝕂 plus en détails, en formalisant un langage impératif nommé IMP (Section 5). Nous expliquons ensuite comment une sémantique 𝕂 est traduite dans Kore qui est une théorie de la Matching Logic (Section 6). Enfin, nous présentons l'outil KaMeLo qui effectue la traduction de Kore vers Dedukti (Section 7).

Dans la suite, les mots-clefs d'un langage ou ce qui est natif dans un langage seront distingués par de la couleur. Le langage de Dedukti se différenciera par une `couleur bleue`, le langage 𝕂 par une `couleur orange`, le langage de Kore par une `couleur rouge`, et le langage de la Matching Logic par une `couleur verte`. Celles-ci facilitent la lecture, mais ne sont pas nécessaires à la compréhension.

2 Le λΠ-calcul modulo théorie

Le λΠ-calcul modulo théorie, abrégé $\lambda\Pi\equiv_{\mathcal{T}}$ par la suite, est un *framework* logique, c'est-à-dire un *framework* permettant de définir des théories, introduit par Cousineau et Dowek [17]. Celui-ci est une extension du λΠ-calcul, avec une notion primitive de calcul définie à l'aide de règles de réécriture [18]. Actuellement, de nombreuses théories ont pu être encodées dans ce framework comme, par exemple, le Calcul des Constructions [10]. La syntaxe, ainsi que les règles de typage[1] qui régissent le $\lambda\Pi\equiv_{\mathcal{T}}$ sont disponibles en figure 1, où le jugement de typage $\Gamma \vdash t : A$ signifie que le terme t a le type A sous le contexte Γ. Le jugement spécifique $\Gamma \vdash A : \mathbf{Type}$ indique que A est un type sous le contexte Γ. Nous considérons également une signature Σ, ainsi qu'un ensemble de règles de réécriture $\mathcal{R}$. Dans ce cadre, toute règle de réécriture $l \hookrightarrow r \in \mathcal{R}$ vérifie $Var(r) \subseteq Var(l)$, où $Var(p)$ est l'ensemble des variables de p, et l'utilisation d'une telle règle de réécriture nécessite que les instanciations $l\sigma$ et $r\sigma$ de ses membres gauche et droit soient toutes deux bien typées, avec le même type ($\Gamma \vdash l\sigma : A$ et $\Gamma \vdash r\sigma : A$ pour un certain type A).

1. D'après l'isomorphisme de Curry-Horward, ces règles constituent également un système de preuve.

Syntaxe	s	$:=$	**Type** | **Kind**	*sorte*
	t	$:=$	$s \mid c \mid x \mid t\ t \mid \lambda(x:t).t \mid \Pi(x:t).t$	*terme*
	Γ	$:=$	$\emptyset \mid \Gamma, x:t$	*contexte*
		avec	c une constante appartenant à Σ,	
			x une variable	

Typage

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

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

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

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

où $s \in \{\textbf{Type}\ ; \textbf{Kind}\}$, $B\{x \backslash a\}$ désigne la substitution de a à x dans B et $\equiv_{\beta\mathcal{R}}$ est la fermeture réflexive, transitive, symétrique et contextuelle de $\equiv$, générée par les règles $\equiv_{reduc}$ et $\equiv_{rule}$.

Figure 1 – Syntaxe et typage du $\lambda\Pi\equiv_{\mathcal{T}}$ avec une signature Σ et des règles de réécriture $\mathcal{R}$

Notons que dans la règle de conversion (conv), la relation d'équivalence dépend non seulement de la β-réduction mais aussi du système de réécriture $\mathcal{R}$. De plus, afin d'avoir la décidabilité du typage, la condition $A \equiv_{\beta\mathcal{R}} B$ dans la règle (conv) doit être décidable, ce qui est assuré lorsque les systèmes de réécriture considérés sont confluents et terminent. Enfin, les contextes peuvent contenir des éléments mal formés et l'ordre des éléments n'a pas d'importance. La bonne formation des éléments du contexte n'est assurée que lors de leur utilisation, dans la règle (var). Cette présentation a été démontrée équivalente aux présentations usuelles dans [19].

3 Un framework pour la logique : Dedukti

Dedukti [9, 2, 3] est un *framework* logique basé sur le $\lambda\Pi\equiv_{\mathcal{T}}$. Ces fondements logiques permettent d'encoder au sein de Dedukti des logiques très expressives comme le Calcul des Constructions Inductives, en les définissant comme des théories du $\lambda\Pi\equiv_{\mathcal{T}}$. Cela permet d'assurer une interopérabilité des preuves entre différents outils formels comme Coq ou PVS (Figure 2).

Dans cette section, nous ne présentons que les fonctionnalités disponibles dans Dedukti sur lesquelles nous nous appuyons dans la suite et qui permettront de faciliter, ultérieurement, les traductions vers d'autres formalismes.

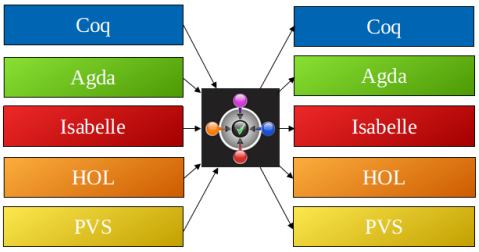

Figure 2 – L'approche de Dedukti, ayant une philosophie similaire à celle de 𝕂, mais pour les preuves

Organisation d'un développement en Dedukti. La notion de module n'existe qu'au niveau d'un fichier. Par conséquent, un fichier peut être vu comme un module, mais il n'est pas possible de déclarer un module à l'intérieur d'un fichier. Cependant, il est possible d'importer un ou plusieurs fichiers dans un même fichier, avec les mots-clef `require` et/ou `open`, voire de le renommer avant de l'utiliser par la suite, avec le mot-clef `as`.

Typage et symboles dans Dedukti. La syntaxe du $\lambda\Pi\equiv_{\mathcal{T}}$ est directement accessible dans Dedukti : `TYPE` (**Kind** n'étant pas accessible à l'utilisateur car inféré par le système), λ (abstraction), Π (produit dépendant), mais aussi $\rightarrow$ qui est utilisé lorsque le produit est non-dépendant. La signature peut être définie à partir de symboles constants (`constant`), c'est-à-dire qui ne peuvent pas être réduits par une règle de réécriture. Si la déclaration d'un symbole est faite avec le mot-clef `symbol` seul, le symbole est dit défini, sans propriété particulière.

Règles de réécriture dans Dedukti. Dans Dedukti, une règle de réécriture s'écrit `rule` $LHS \hookrightarrow RHS$ dans laquelle les variables sont notées `$x`, `$y`, etc. Il est possible d'y utiliser un joker (`_`) à gauche lorsqu'une variable n'est pas utilisée dans la partie droite. Pour des raisons d'efficacité, il est possible de déclarer plusieurs règles à la fois à l'aide du mot-clef `with`. Les règles de réécriture autorisent l'ordre supérieur, peuvent être non linéaires et ne s'appliquent pas forcément en tête de terme, mais ne sont pas conditionnelles. Toutes les règles de réécriture définies dans Dedukti constituent un unique ensemble : il n'y a pas de notion de priorité entre ces différentes règles, lorsque nous cherchons la prochaine règle qui s'applique.

4 La Matching Logic

Le framework 𝕂 a d'abord été développé pour faciliter l'écriture de sémantiques, avec l'objectif de ne garder que le meilleur des différents styles de définition sémantique. Ensuite, la recherche de fondements théoriques pour ce framework a donné lieu à différentes logiques qui s'améliorent les unes les autres [26, 28, 16, 15]. Ici, nous nous inspirons de la présentation de la Matching Logic faite à ISR 2021 [5] et de l'article [14]. Ainsi une définition sémantique en 𝕂 peut être considérée comme une théorie de la Matching Logic. D'autres formalisations de la sémantique de 𝕂 ont été proposées dans la littérature. Par exemple, Li et Gunter proposent une formalisation de la sémantique de 𝕂 en Isabelle/HOL [21, 22]. De plus, il existe des travaux sur la définition de la sémantique de 𝕂 en 𝕂 [1], et plus récemment, la Matching Logic a été formalisée à l'aide de Metamath [13].

Syntaxe. La Matching Logic est une logique non typée du 1er ordre qui suit la philosophie « terms as formulae », c'est-à-dire telle qu'il n'y a pas de distinction entre les termes et les formules. En effet, cette logique manipule des *patterns*. Soit Σ un ensemble de symboles constants, nommé également signature. L'ensemble des Σ-patterns est défini à l'aide de huit constructeurs : les variables d'élément (`x`), les variables d'ensemble (`X`), les symboles (σ), l'application ($\varphi_1\ \ \varphi_2$), les connecteurs propositionnels ($\bot$ et $\rightarrow$), le quantificateur existentiel ($\exists x.\varphi$) et l'opérateur de plus petit point fixe ($\mu X.\varphi$, où il n'y a pas d'occurrences négatives de X dans φ).

Notations. Il est possible d'introduire quelques notations, sans étendre l'expressivité de la logique. Nous avons regroupé différentes notations usuelles à la figure 3, en notant $\varphi[\psi/v]$ la substitution du pattern ψ à v, une variable d'élément ou d'ensemble, dans le pattern φ.

Sémantique. Le nom de la logique est étroitement lié à la notion de *pattern-matching*, d'où le vocabulaire et la sémantique associés. Intuitivement, un pattern φ est interprété par l'ensemble des éléments qu'il *matche*. La sémantique associée à chaque constructeur de pattern est donc étroitement liée à des opérations usuelles sur les ensembles. Nous notons Σ-modèle, un triplet $(M, @_M, \{\sigma_M\}_{\{\sigma\in\Sigma\}})$ où M est un ensemble support non vide, $@_M$ est l'interprétation du pattern applicatif, de type $M \times M \rightarrow \mathcal{P}(M)$ et σ_M est l'interprétation des symboles telle que

Constructeur	Sémantique
$\mathtt{x}$	$\lvert x\rvert_{M,\rho} = \{\rho(x)\}$
$\mathtt{X}$	$\lvert X\rvert_{M,\rho} = \rho(X)$
σ	$\lvert\sigma\rvert_{M,\rho} = \sigma_M$
$\varphi_1\ \varphi_2$	$\lvert\varphi_1\ \varphi_2\rvert_{M,\rho} = \bigcup_{\substack{a_1\in\lvert\varphi_1\rvert_{M,\rho}\\ a_2\in\lvert\varphi_2\rvert_{M,\rho}}} a_1 \mathbin{@} a_2$
$\bot$	$\lvert\bot\rvert_{M,\rho} = \emptyset$
$\to$	$\lvert\varphi_1 \to \varphi_2\rvert_{M,\rho} = M \setminus (\lvert\varphi_1\rvert_{M,\rho} \setminus \lvert\varphi_2\rvert_{M,\rho})$
$\exists x.\varphi$	$\lvert\exists x.\varphi\rvert_{M,\rho} = \bigcup_{a\in M} \lvert\varphi\rvert_{M,\rho[a/x]}$
$\mu X.\varphi$	$\lvert\mu X.\varphi\rvert_{M,\rho} = \mathbf{lfp}(A \mapsto \lvert\varphi\rvert_{M,\rho[a/X]})$

Notation	Définition
$\neg\varphi_1$	$\varphi_1 \to \bot$
$\top$	$\neg\bot$
$\varphi_1 \vee \varphi_2$	$\neg\varphi_1 \to \varphi_2$
$\varphi_1 \wedge \varphi_2$	$\neg(\neg\varphi_1 \vee \neg\varphi_2)$
$\varphi_1 \leftrightarrow \varphi_2$	$(\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$
$\forall x.\varphi$	$\neg\exists x.\neg\varphi$
$\nu X.\varphi$	$\neg\mu X.\neg\varphi[\neg X/X]$

Figure 3 – Syntaxe, notation et sémantique de la Matching Logic

$\forall\sigma \in \Sigma.\ \sigma_M \subseteq M$. Cette définition montre que la Matching Logic a une interprétation ensembliste, alors que la logique des prédicats a une interprétation fonctionnelle, ce qui en fait un cas particulier, en considérant qu'il y a une bijection entre a et $\{a\}$. Les sémantiques, notées $\lvert\ .\ \rvert_{M,\rho}$, pour les constructeurs sont données à la figure 3, où ρ une valuation telle que $\rho(x) \in M$ pour toute variable d'élément x et $\rho(X) \subseteq M$ pour toute variable d'ensemble X. et **lfp** est une fonction de plus petit point fixe. Les sémantiques pour les notations peuvent être retrouvées par le calcul.

Théories. Tout comme le $\lambda\Pi\equiv_{\mathcal{T}}$, la Matching Logic est une logique permettant de raisonner modulo une théorie. Il est donc tout à fait possible d'ajouter à la Matching Logic la théorie de l'égalité à l'aide du symbole $\lceil\_\rceil$ (*definedness symbol*), la théorie des sortes à l'aide du symbole $[\![\_]\!]$ (*inhabitant symbol*) ou encore la théorie de la réécriture à l'aide de la sorte *State* et du symbole $\bullet$ (*one-path next symbol*). La sémantique de ce dernier symbole repose sur le fait que les règles de réécriture définissent une relation binaire $\mathcal{R}$ entre des états, qui sont de type *State*, c'est-à-dire qu'elles définissent un système de transition.
Les définitions et sémantiques nécessaires sont présentées en figure 4, où $[\![s]\!]_M$ est l'ensemble support de s dans M.

Théorie	Symbole	Sémantique	Axiome
Égalité	$\lceil\varphi\rceil$	$\lvert\lceil\varphi\rceil\rvert = M$ *si* $\lvert\varphi\rvert \neq \emptyset$, $\lvert\lceil\varphi\rceil\rvert = \emptyset$ *sinon*	$\forall x.\ \lceil x\rceil$
Sortes	$[\![s]\!]$	$\lvert[\![s]\!]\rvert = [\![s]\!]_M$	
Réécriture	*State*	*State* est une sorte donc $\lvert[\![State]\!]\rvert = [\![State]\!]_M$	
	$\bullet\varphi$	$\lvert\bullet\varphi\rvert = \{s \in [\![State]\!] \mid \exists t \in \lvert\varphi\rvert$ *tel que* $s \hookrightarrow t \in \mathcal{R}\}$	

$\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil \quad \varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \to \varphi_2\rfloor \quad \varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor \quad x \in \varphi \equiv \lceil x \wedge \varphi\rceil \quad \neg_s\varphi \equiv (\neg\varphi) \wedge [\![s]\!]$

$\forall x : s.\varphi \equiv \forall x.(x \in [\![s]\!]) \to \varphi \quad \varphi_1 \hookrightarrow \varphi_2 \equiv \varphi_1 \to \bullet\varphi_2 \quad \diamond\varphi \equiv \mu X.\varphi \vee \bullet X \quad \varphi_1 \hookrightarrow^* \varphi_2 \equiv \varphi_1 \to \diamond\varphi_2$

Figure 4 – Extensions de la Matching Logic

Ces trois théories constituent la théorie nommée Kore, où *State* est le type des configurations.

Système de preuve. Le système de preuve de la Matching Logic est présenté en figure 5, où C, C_1, C_2 sont des contextes applicatifs, respectant la grammaire $C ::= \Box \mid C\ \varphi \mid \varphi\ C$.
A travers l'axiome (Prop 3), nous constatons que la Matching Logic est une logique classique.

FOL Reasoning

$$\frac{}{\varphi \to (\psi \to \varphi)}\ \text{(Prop 1)}$$

$$\frac{}{(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))}\ \text{(Prop 2)}$$

$$\frac{}{((\varphi \to \bot) \to \bot) \to \varphi}\ \text{(Prop 3)}$$

$$\frac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}\ \text{(Modus Ponens)}$$

$$\frac{}{\varphi[y/x] \to \exists x.\varphi}\ (\exists\text{-Quantifier})$$

$$\frac{\varphi_1 \to \varphi_2 \quad (when\ x \notin FV(\varphi_2))}{(\exists x.\varphi_1) \to \varphi_2}\ (\exists\text{-Generalization})$$

Technical rules

$$\frac{}{\exists x.x}\ \text{(Existence)}$$

$$\frac{}{\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])}\ \text{(Singleton)}$$

Fixpoint Reasoning

$$\frac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]}\ \text{(Framing)}$$

$$\frac{\varphi}{\varphi[\psi/X]}\ \text{(Set Variable Substitution)}$$

$$\frac{}{\varphi[(\mu X.\varphi/X)] \to \mu X.\varphi}\ \text{(PreFixpoint)}$$

$$\frac{\varphi[\psi/X] \to \psi}{\mu X.\varphi \to \psi}\ \text{(Knaster-Tarski)}$$

Frame Reasoning

$$\frac{}{C[\bot] \to \bot}\ (Propagation_\bot)$$

$$\frac{}{C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]}\ (Propagation_\vee)$$

$$\frac{(when\ x \notin FV(C))}{C[\exists x.\varphi] \to \exists x.C[\varphi]}\ (Propagation_\exists)$$

Figure 5 – Système de preuve de la Matching Logic

5 Un framework pour la sémantique : 𝕂

Le framework 𝕂 [6] permet de définir des sémantiques formelles de langages de programmation et de générer automatiquement des outils pour un langage à partir de sa sémantique formelle (Figure 6). Créé par Grigore Rosu en 2003 à l'Université de Champaign-Urbana (USA - Illinois), 𝕂 est aujourd'hui maintenu par l'entreprise Runtime Verification.
Actuellement, les principaux backends sont le backend LLVM pour l'exécution, et le backend Haskell pour l'exécution symbolique.

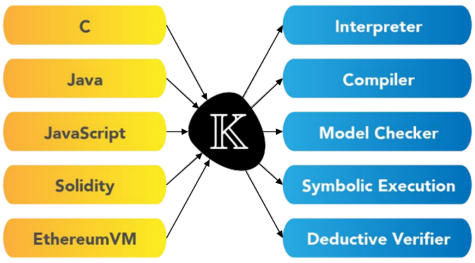

Figure 6 – L'approche de 𝕂

Il est possible de considérer 𝕂 comme une sur-couche orientée notation au-dessus de la logique de réécriture, spécialisée et optimisée pour écrire les définitions de langages et de modèles de programmation complexes. Pour illustrer les fonctionnalités de 𝕂, nous allons montrer comment écrire la sémantique d'un petit langage impératif que nous nommons IMP. Pour une présentation plus complète, le lecteur peut se rapporter à [25].

Organisation d'un développement en 𝕂. Contrairement à Dedukti, la notion de fichier est disjointe de la notion de module puisqu'un fichier peut contenir plusieurs modules (`module`/`endmodule`). De plus, il est possible d'importer des fichiers dans un autre fichier (`requires`), ou encore d'importer un ou plusieurs modules dans un autre module (`imports`).

5.1 Définir la syntaxe d'un langage dans 𝕂

Définir la syntaxe de IMP dans 𝕂 s'apparente à l'écriture d'une grammaire BNF, comme le montre la partie gauche de la colonne « Syntaxe » de la figure 7. La partie droite de cette même colonne contient des attributs, que nous expliquons au fur et à mesure. Dans la suite, un symbole terminal désignera un symbole, pouvant être *mixfix*, écrit entre guillemets, comme par exemple _"+"_, ou encore "`while`"_"`do`"_. Tout ce qui n'est pas entre guillemets sera donc un symbole non terminal.
Pour rendre la syntaxe précédente analysable, il est possible d'utiliser des attributs permettant de préciser la précédence des symboles (`prec(nb)`), ainsi que l'associativité (`left`, `right`, `non-assoc`), ou encore d'ajouter des parenthèses au langage, à l'aide de l'attribut `bracket`. Il sera alors possible de générer un analyseur syntaxique pour IMP.

5.2 Définir la sémantique d'un langage dans 𝕂

Pour définir la sémantique dynamique associée à chaque élément de la syntaxe, il est possible d'utiliser ce qui a déjà été défini dans la bibliothèque standard, mais aussi des configurations, des règles de réécriture et des attributs. La sémantique de IMP est définie à la colonne « Sémantique » de la figure 7.

Syntaxe		Sémantique	n°
`syntax` **AExp** ::= `Int` | `Id`		`rule` $\langle\langle\ x \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env}\rangle$ `=>` $\langle\langle\ i \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env}\rangle$	1
| **AExp** "/" **AExp**	[`left`, `strict`]	`rule` i_1/i_2 `=>` $i_1\ /_{Int}\ i_2$ `requires` $i_2 \neq_{Int} 0$	2
> **AExp** "+" **AExp**	[`left`, `strict`]	`rule` $i_1 + i_2$ `=>` $i_1\ +_{Int}\ i_2$	3
| "(" **AExp** ")"	[`bracket`]		
`syntax` **BExp** ::= `Bool`			
| **AExp** "<" **AExp**	[`seqstrict`]	`rule` $i_1 < i_2$ `=>` $i_1 <_{Int} i_2$	4
| "`not`" **BExp**	[`strict`]	`rule not` t `=>` $\neg_{Bool}\ t$	5
> **BExp** "`and`" **BExp**	[`left`, `strict(1)`]	`rule` $true$ `and` b `=>` b	6
| "(" **BExp** ")"	[`bracket`]	`rule` $false$ `and` _ `=>` $false$	7
`syntax` **Stmt** ::= "{" "}"		`rule { } =>` .	8
| "{" **Stmt** "}"		`rule { }` s `=>` s	9
| `Id` "=" **AExp** ";"	[`strict(2)`]	`rule` $\langle\langle\ x = i; \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto \_\ \rangle_{env}\rangle$ `=>` $\langle\langle\ s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env}\rangle$	10
| "`if`" **BExp** "`then`" **Stmt** "`else`" **Stmt**	[`strict(1)`]	`rule if` $true$ `then` s `else` _ `=>` s	11
		`rule if` $false$ `then` _ `else` s `=>` s	12
| "`while`" **BExp** "`do`" **Stmt**		`rule while` b `do` s `=>` `if` b `then` {s `while` b `do` s} `else { }`	13
> **Stmt Stmt**	[`left`]	`rule` $s_1\ s_2$ `=>` $s_1 \curvearrowright s_2$	14
`syntax` **Ids** ::= `List{Id, ","}`		`rule var` $.Ids$ `; s =>` s	15
`syntax` **Pgm** ::= "`var`" **Ids** ";" **Stmt**		`rule var` x, xl`; s` $\rangle_k\ \langle\ m\ \rangle_{env}\rangle$ `=>` $\langle\langle$ `var` xl`; s` $\rangle_k\ \langle\ m\ \ x \mapsto 0\ \rangle_{env}\rangle$ `requires` $x \notin m$	16

Figure 7 – Syntaxe et sémantique de notre langage IMP

Ce qui est natif dans 𝕂, la bibliothèque standard. La bibliothèque standard de 𝕂[2] définit des sortes courantes comme `Id`, `Int` ou encore `Bool` (avec les constantes $true$ et $false$), mais aussi des opérateurs usuels suffixés par une sorte, comme l'égalité sur les entiers (`==Int`), la diségalité sur les entiers (`=/=Int`) ou encore la conjonction booléenne (`andBool`). De plus, cette bibliothèque définit les sortes particulières pour les listes (`List`), les ensembles (`Set`), les multi-ensembles (`Bag`) et les dictionnaires (`Map`). Par convention, les éléments neutres pour la concaténation commencent par `.` dans 𝕂. Par exemple, `.Map` correspond à un dictionnaire vide, tandis que `.Ids` correspond à une liste vide de type `Ids`.

2. Disponible ici : https://kframework.org/k-distribution/include/kframework/builtin/.

Pour finir, la sorte `K` indique qu'un élément de type `K` est calculatoire : tout ce qui est défini dans 𝕂 est de type `K`. Les *K computations* sont un cas particulier de contenu calculatoire, donc de type `K`, particulièrement utiles pour définir des stratégies d'évaluation. Formellement, une K computation est une liste, potentiellement imbriquée et ayant pour constructeurs $\curvearrowright$ et `.`, de calculs à réaliser de manière séquentielle. Par exemple, la K computation x = 10 ; $\curvearrowright$ y = 42 ; modélise le fait que nous affectons d'abord une valeur à la variable x avant de modifier y.

Configurations. Formellement, une configuration est un multi-ensemble de cellules étiquetées, potentiellement imbriquées. Plus concrètement, une configuration contient toutes les informations sémantiques nécessaires pour exécuter un programme : nous pouvons voir une configuration comme un état du programme, qui stocke par exemple le programme à exécuter, les valeurs courantes des variables ou encore la pile. Pour le langage IMP, nous nous contentons d'une configuration à deux cellules, l'une étiquetée par k contenant le programme à exécuter, l'autre étiquetée par *env* contenant les valeurs courantes des variables. Pour définir une telle configuration en 𝕂, il suffit de spécifier la configuration initiale souhaitée, comme par exemple `configuration` <k>`$PGM` :**Pgm**</k> <env>`.Map`</env> qui indique que le programme à exécuter (`$PGM`) doit être mis dans la cellule k et que l'environnement est initialement vide. Dans la suite de cet article, et en accord avec les notations usuelles utilisées dans les articles sur 𝕂, nous allégeons la notation pour décrire une configuration, comme $\langle\langle\ x = 10;\ \rangle_k\ \langle\ x \mapsto 0\ \rangle_{env} \rangle$.

Règles de réécriture. Dans 𝕂, une règle de réécriture, notée `rule` *LHS* `=>` *RHS*, est du 1er ordre et s'applique sur des configurations. Par exemple, la règle suivante signifie que la variable x est évaluée à i puisque la valeur associée à x dans l'environnement, ici la cellule *env*, est i : `rule` $\langle\langle\ x \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env} \rangle$ `=>` $\langle\langle\ i \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env} \rangle$ (règle n°1 - Figure 7).

Comme dans Dedukti, les règles de réécriture de 𝕂 peuvent être non-linéaires, et ne s'appliquent pas forcément en tête de terme, c'est-à-dire sur une configuration entière. C'est le cas de la règle n°3 (Figure 7) qui donne la sémantique de l'opérateur "+". Dans ce cas de figure, le reste de la configuration est inféré par 𝕂 lors de la compilation en Kore. Cette écriture permet de faire des raisonnements locaux, mais permet surtout d'être plus modulaire. Si l'utilisateur souhaite qu'une règle de réécriture puisse être appliquée n'importe où comme dans Dedukti, il lui suffit d'ajouter l'attribut `anywhere` à cette règle. De plus, ce qui n'est pas utilisé dans le membre droit peut être omis à l'aide d'un joker, comme aux règles n°7, 10 et 12 (Figure 7).

Contrairement à Dedukti, une règle de réécriture peut être conditionnelle dans 𝕂, comme le montre la règle n°2 (Figure 7) qui définit la sémantique de la division.

Définir une stratégie d'évaluation. Il existe deux attributs pour définir une stratégie d'évaluation, c'est-à-dire préciser l'ordre dans lequel les sous-expressions sont évaluées : `strict`, si elle s'effectue de manière non déterministe, et `seqstrict` si elle s'effectue de manière déterministe, de gauche à droite par défaut. Il est également possible de restreindre la liste des arguments qui doivent être évalués avant l'évaluation de l'instruction globale en donnant une liste de nombres, chacun indiquant la position d'un symbole non terminal. Par exemple, pour définir un « et paresseux », nous pouvons utiliser l'attribut `strict(1)`, ainsi que les deux règles suivantes : `rule` *true* `and` b `=>` b et `rule` *false* `and _ =>` *false* (règles n°11 et n°12 - Figure 7).

Enfin, pour spécifier des stratégies d'évaluation plus complexes, il est également possible d'utiliser des contextes (`context`), comme cela se fait classiquement, mais également des squelettes de contextes (`context alias`), permettant de générer automatiquement des contextes, plutôt que d'écrire systématiquement toujours les mêmes contextes. Nous verrons à la section 6.2 une dernière méthode pour définir une stratégie d'évaluation, à l'aide des K computations.

D'autres attributs. Il existe de très nombreux attributs [3], mais à notre connaissance, il n'existe pas de liste d'attributs exhaustive totalement documentée. Nous présentons, ci-dessous, les derniers attributs qui nous intéressent dans le cadre de ce travail.

Tout symbole est soit un symbole de constructeur (`constructor`), soit un symbole de fonction (`function`), mais pas les deux. Par défaut, un symbole est un symbole de constructeur. Nous verrons que la différence entre ces deux attributs influence la traduction de 𝕂 vers Kore, comme le fait que le reste de la configuration est inféré uniquement si le symbole de tête a l'attribut `constructor`. De plus, il est possible de préciser qu'un symbole est injectif (`injective`) ou encore total (`functional`). Enfin, un symbole peut être manipulé modulo associativité (`assoc`), commutativité (`comm`), unité [4] (`unit`), idempotence (`idem`). Ces quatre derniers attributs permettent de faire de la réécriture modulo ACUI.

Les règles de réécriture, par défaut, ne s'appliquent pas dans un ordre particulier. Cependant, l'attribut `owise` indique qu'une règle ne s'applique que lorsque aucune autre ne s'applique, et l'attribut `priority(`*`nb`*`)` permet de préciser l'ordre d'application des règles.

6 De 𝕂 à la Matching Logic : Kore

Le langage Kore est un langage intermédiaire entre le langage de 𝕂 et le langage de la Matching Logic, permettant ainsi de minimiser l'écart entre le langage qui décrit la sémantique, celui de 𝕂, et le langage qui constitue les fondements logiques de 𝕂, la Matching Logic. La compilation d'une définition sémantique en 𝕂 génère un fichier au format Kore. Notre traduction de 𝕂 en Dedukti, présentée à la section 7, utilise ce format Kore intermédiaire, et non le fichier source écrit en 𝕂. Nous supposons donc correcte la traduction de 𝕂 vers Kore. Cette section explique comment s'effectue cette traduction entre 𝕂 et la Matching Logic, explications qui résultent d'une étude des fichiers Kore générés et de discussions avec des membres de l'équipe de développement, puisque aucun document formalisant cette étape n'a été publié. La formalisation partielle de cette traduction est disponible à la figure 8.

Organisation d'un développement en Kore. Un fichier Kore est composé d'une succession de modules (`module`/`endmodule`), commençant éventuellement par des déclarations d'importation (`import`), elles-mêmes suivies de déclarations commençant par l'un des mot-clefs suivants : `sort`, `hooked-sort`, `symbol`, `hooked-symbol`, `alias`/`where`, `axiom`. Quelle que soit la hiérarchie des fichiers qui constitue la sémantique d'un langage, le processus de compilation vers Kore va tout fusionner dans un unique module. Quatre modules [5] sont également importés par défaut lors de la traduction de 𝕂 vers Kore : `BASIC-K` qui contient les sortes `SortK` et `SortKItem`, correspondant aux sortes `K` et `KItem` dans 𝕂 ; `KSEQ` qui contient les constructeurs des K computations `kseq` et `dotk`, notés $\curvearrowright$ et `.` dans 𝕂 ; `INJ` qui introduit le symbole `inj` pour modéliser des injections et enfin, `K` qui contient principalement les modules précédents. Le module `K` est importé dans le module qui contient la sémantique.

Notion de typage. Lors de la traduction de 𝕂 à Kore, tous les types sont précisés à l'aide d'injections (`inj`), informations de typage que nous utilisons parfois, comme à la section 7.3.

6.1 Traduction de la syntaxe dans Kore

La traduction de 𝕂 vers Kore procède de deux manières différentes selon la forme de `B` dans une déclaration `syntax A ::= B`, comme le montre la figure 8 [6]. Si `B` n'a pas de symbole

3. Une liste non exhaustive est présente à la fin de cette page https://kframework.org/USER_MANUAL/.
4. On parle aussi d'*identité*, c'est-à-dire modulo les éléments neutres.
5. Les modules `BASIC-K`, `KSEQ`, `INJ` et `K` sont disponibles ici : https://github.com/kframework/kore/blob/master/src/main/kore/prelude.kore, ou au chemin "/usr/include/kframework/builtin/prelude.md".
6. Dans cette figure, de nombreuses déclarations introduisent des accolades laissées vides. En effet, de très rares cas nécessitent de traduire les paramètres de type mis entre accolades.

Module et importation (I_j^h est de la forme `imports` I_j^h [ ]), $k \geq 1$ et $\forall j.\ n_j \geq 0$

|| `module` I_1^1 ... $I_{n_1}^1$ M_1 `endmodule` ... `module` I_1^k ... $I_{n_k}^k$ M_k `endmodule` ||$_{\texttt{kore}}$ =
  || Prelude ||$_{\texttt{aux}}$ `module import K` [ ] || M_1 ||$_{\texttt{kore}}$ ... || M_k ||$_{\texttt{kore}}$ `endmodule` [ ]
|| Prelude ||$_{\texttt{aux}}$ = Modules `BASIC-K`, `KSEQ`, `INJ` et `K`

Syntaxe (A est un symbole non terminal)

|| `syntax` A : := B [Attr] ||$_{\texttt{kore}}$ =
  || `sort` A{} [ ] ||$_{\texttt{aux}}$
  || `sort` B{} [ ] ||$_{\texttt{aux}}$
  `axiom` $\varphi_{B \subseteq A}$ [`subsort`]

Si B n'a pas de symbole terminal.

|| `syntax` A : := B [Attr] ||$_{\texttt{kore}}$ =
  || `sort` A{} [ ] ||$_{\texttt{aux}}$
  || `symbol` B{}(T_1, ..., T_m) : A [Attr] ||$_{\texttt{aux}}$

Si B est de la forme $X_1\ X_2$ ... X_n avec $n \geq 1$ et X_i un symbole, et $\forall i \in [1..m].\exists j.\ X_j = T_i$ avec T_i un symbole non terminal.

|| `sort` A{} [ ] ||$_{\texttt{aux}}$ =
  `sort` A{} [ ]
  || $Proj_A$ et $Pred_A$ ||$_{\texttt{aux}}$

Si A n'est pas dans la bibliothèque de $\mathbb{K}$.

|| `sort` A{} [ ] ||$_{\texttt{aux}}$ =
  `hooked-sort` A{} [ ]
  || $Proj_A$ et $Pred_A$ ||$_{\texttt{aux}}$

Si A est dans la bibliothèque de $\mathbb{K}$.

|| $Proj_A$ et $Pred_A$ ||$_{\texttt{aux}}$ =
  || `symbol` project$A${}(...) : A [`projection`] ||$_{\texttt{aux}}$
  || `symbol` is$A${}(...) : `SortBool` [`predicate`] ||$_{\texttt{aux}}$

Si A $\neq$ `SortKConfigVar`.

|| `symbol` B{}(T_1, ..., T_m) : A [x, ...] ||$_{\texttt{aux}}$ =
  `symbol` B {}(T_1, ..., T_m) : A [x, ...]
  `axiom` φ_x [x, ...]

Si x $\in$ {`assoc`, `comm`, `unit`, `idem` `functional`, `constructor`, `injective`, `projection`, `predicate`, `initializer` }.

Configuration

|| `configuration` C ||$_{\texttt{kore}}$ =
  || `sort` Sort$cell${} [ ] ||$_{\texttt{aux}}$
  || `symbol` $cell${} [`cell`, ...] ||$_{\texttt{aux}}$
  || `symbol` Init$cell${} [`initializer`] ||$_{\texttt{aux}}$
  || `sort SortKConfigVar` {} [ ] ||$_{\texttt{aux}}$

Où $cell \in C$.

Si `$PGM` est une valeur d'initialisation.

Règle de réécriture

|| `rule` LHS `=>` RHS `requires` $Cond$ [$Attr$] ||$_{\texttt{kore}}$ =
  `alias` A{}(T_1, ..., T_n) : T_e `where` A{}($N_1 : T_1$, ..., $N_n : T_n$)
    := $\overline{Cond}^{\varphi}\ \wedge\ mgCONF(\overline{LHS}^{\varphi})$ [ ]
  `axiom` A{}($N_1 : T_1$, ..., $N_n : T_n$) $\hookrightarrow$
      $\top\ \wedge\ mgCONF(\overline{RHS}^{\varphi})$) [$Attr$]

Si le symbole de tête a l'attribut `constructor`.
Où A est un alias pour LHS, $T_1, ..., T_n$ correspondent aux types des variables de LHS, et T_e est le type de LHS.

|| `rule` LHS `=>` RHS `requires` $Cond$ [$Attr$] ||$_{\texttt{kore}}$ =
  `axiom` $\overline{Cond}^{\varphi}\ \wedge\ Data\ \rightarrow$
      $(\overline{LHS}^{\varphi}\ =\ (\overline{RHS}^{\varphi}\ \wedge\ \top))$ [$Attr$]

Si le symbole de tête a l'attribut `function`.
Où $Data$ décrit des associations entre des variables et des patterns.

Stratégie d'évaluation

|| `syntax` A : := B [`strict`, Attr] ||$_{\texttt{kore}}$
|| `syntax` A : := B [`seqstrict`, Attr] ||$_{\texttt{kore}}$
|| `context` Co ||$_{\texttt{kore}}$
|| `context alias` CA ||$_{\texttt{kore}}$

Voir section 6.2 pour des exemples de transformation en K computations.

Figure 8 – Règles de traduction de $\mathbb{K}$ vers Kore

terminal, comme dans `syntax AExp ::= Int`, alors il est généré une sorte `B` ainsi qu'un *axiome de sous-typage* ayant l'attribut `subsort`, précisant que `B` est une sous-sorte de `A`, noté $\varphi_{B \subseteq A}$. Mais si `B` a au moins un symbole terminal, comme dans `syntax AExp ::= AExp "-" AExp`, alors est introduit un symbole `B` qui conserve les attributs précisés par l'utilisateur. De plus, pour chaque sorte générée, sont ajoutés un symbole de projection [7], ayant l'attribut `projection`, permettant de projeter une entité de type `SortK` vers un autre sous-type, ainsi qu'un symbole de prédicat, comme `isBool`, ayant l'attribut `predicate`, indiquant qu'une entité est du type associé au symbole de prédicat.

6.2 Traduction de la sémantique dans Kore

Configurations. La configuration initialement déclarée dans 𝕂 est découpée en plusieurs morceaux, un morceau par cellule. Chaque cellule [8] génère une sorte pour typer la cellule, ainsi qu'un symbole avec au moins l'attribut `cell` pour la représenter. De plus, la configuration initiale précise la valeur d'initialisation de chaque cellule, ce qui génère un symbole avec l'attribut `initializer`, correspondant à cette valeur d'initialisation.

Règles de réécriture. La traduction vers Kore d'une règle de réécriture de la forme `rule` LHS `=>` RHS `requires` C_{pre} $[Attr]$ génère un alias (`alias`/`where`) du LHS utilisé dans un axiome commençant par $\hookrightarrow$, si le symbole de tête possède l'attribut `constructor`. Cet axiome encode la règle de réécriture de la manière suivante : $C_{pre} \wedge LHS \hookrightarrow RHS$ [9]. Si la clause `requires` C_{pre} est omise, alors $\overline{C_{pre}}^{\varphi}$ devient $\top$ lors de la traduction. Si le symbole de tête possède l'attribut `function`, l'encodage est similaire mais l'axiome commence par $\rightarrow$. La formalisation de ces traductions sont disponibles à la figure 8, où la fonction $\overline{f}^{\varphi}$ transforme f en un pattern de la Matching Logic, et la fonction $mgCONF$ calcule le reste de la configuration. Si la règle possède l'attribut `anywhere`, la fonction $mgCONF$ est l'identité.

Stratégies d'évaluation. Nous avons vu en section 5.2 trois manières différentes de définir des stratégies d'évaluation dans 𝕂 : les attributs `strict` et `seqstrict`, les contextes et les contextes alias. Lors de la traduction vers Kore, les attributs `strict` et `seqstrict`, mais aussi les contextes et les contextes alias sont traduits en utilisant les K computations et des *freezers*. Intuitivement, un freezer permet de « geler » le reste de la K computation, qui devient non-modifiable, en attendant que la tête de la K computation soit évaluée. Par la suite, nous notons ($\boldsymbol{\curlywedge}^{nb}_{sym}$ arg) un freezer où sym est un symbole, nb le numéro de l'argument dont nous attendons la valeur, et arg la liste des autres arguments. Par exemple, pour calculer (4 + 8) - 10, nous construisons la liste $4 + 8 \curvearrowright (\boldsymbol{\curlywedge}^{1}_{\_}\ 10)$ puisque nous souhaitons d'abord évaluer ce qui se trouve entre parenthèses. Nous obtenons ensuite $12 \curvearrowright (\boldsymbol{\curlywedge}^{1}_{\_}\ 10)$ car nous commençons d'abord par calculer le premier élément de la liste, puis nous « dégelons » le calcul pour obtenir 12 - 10, et enfin 2. Cette notion est inspirée des contextes d'évaluation $C[v]$, et des continuations $v \curvearrowright C$ (passer v à la continuation C).

Dans le cas du « et paresseux » du langage IMP, l'attribut `strict(1)` génère la règle de réécriture `rule` E_1 `and` E_2 `=>` $E_1 \curvearrowright (\boldsymbol{\curlywedge}^{1}_{\texttt{and}}\ E_2)$ `requires` $E_1 \notin$ `KResult`, mais aussi la règle symétrique `rule` $E_1 \curvearrowright (\boldsymbol{\curlywedge}^{1}_{\texttt{and}}\ E_2)$ `=>` E_1 `and` E_2 `requires` $E_1 \in$ `KResult`. La sorte `KResult` est une sous-sorte de `K` qui permet de distinguer les valeurs (finales) des expressions. Pour IMP, nous

7. Sauf pour `SortKConfigVar`, la sorte des variables de configuration de 𝕂, comme la variable `$PGM`.

8. Certaines cellules sont ajoutées par rapport à celles précisées par l'utilisateur, comme la cellule `<generatedTop>` qui encapsule toutes les autres cellules. Le processus décrit est le même, outre que l'attribut `cell` devient `cellFragment`, et l'attribut `initializer` devient `cellOptAbsent`, puisque cette cellule n'a pas de valeur initiale.

9. Les propriétés autorisées par le KProver sont de la forme `rule` LHS `=>` RHS `requires` C_{pre} `ensures` C_{post} et sont encodées ainsi : $C_{pre} \wedge LHS \hookrightarrow C_{post} \wedge RHS$. Dans le cadre de cet article, C_{post} vaut donc $\top$, comme le montre la figure 8.

définissons cette sorte ainsi `syntax KResult ::= Int | Bool`, c'est-à-dire qu'une valeur finale est soit un entier, soit un booléen. Ainsi, la première règle force l'évaluation de E_1 si E_1 n'est pas une valeur, tandis que la deuxième règle simplifie la K computation puisque son élément de tête est une valeur. La condition $E_1 \in$ `KResult` devient `isKResult` E_1 dans le format Kore. D'autres exemples de traduction des attributs définissant une stratégie d'évaluation utilisés à la figure 7 sont disponibles à la figure 9.

Règles générées par l'attribut `strict`	Règles générées par l'attribut `seqstrict`
`rule` $E_1 + E_2$ `=>` $E_1 \curvearrowright (\circledast^1_+ E_2)$ `requires` $E_1 \notin$ `KResult`	`rule` $E_1 < E_2$ `=>` $E_1 \curvearrowright (\circledast^1_< E_2)$ `requires` $E_1 \notin$ `KResult`
`rule` $E_1 \curvearrowright (\circledast^1_+ E_2)$ `=>` $E_1 + E_2$ `requires` $E_1 \in$ `KResult`	`rule` $E_1 \curvearrowright (\circledast^1_< E_2)$ `=>` $E_1 < E_2$ `requires` $E_1 \in$ `KResult`
`rule` $E_1 + E_2$ `=>` $E_2 \curvearrowright (\circledast^2_+ E_1)$ `requires` $E_2 \notin$ `KResult`	`rule` $E_1 < E_2$ `=>` $E_2 \curvearrowright (\circledast^2_< E_1)$ `requires` $E_2 \notin$ `KResult` $\wedge$ $E_1 \in$ `KResult`
`rule` $E_2 \curvearrowright (\circledast^2_+ E_1)$ `=>` $E_1 + E_2$ `requires` $E_2 \in$ `KResult`	`rule` $E_2 \curvearrowright (\circledast^2_< E_1)$ `=>` $E_1 < E_2$ `requires` $E_2 \in$ `KResult`

Figure 9 – Traduction des attributs `strict` et `seqstrict` dans Kore

Attributs. Les attributs initialement présents dans le fichier 𝕂 sont *a priori* recopiés dans le fichier Kore. Certains attributs pouvant être associés à un symbole génèrent également un axiome, comme le montre la figure 8. Deux cas particuliers existent. Pour l'attribut `predicate`, deux axiomes sont générés au lieu d'un seul, l'un précisant la sémantique quand le prédicat se réduit à faux, l'autre dans le cas vrai. Pour l'attribut `constructor`, un *axiome d'injectivité* est généré pour chaque symbole ayant cet attribut, ainsi que deux axiomes pour chaque ensemble de constructeurs construisant une valeur d'un même type : un *axiome de non-recouvrement*, précisant que deux valeurs d'un certain type commençant par des constructeurs différents sont différentes ; et un *axiome d'exhaustivité*, précisant que nous ne pouvons construire des valeurs d'un certain type qu'avec les constructeurs associés à ce type.

Tout axiome généré est la formalisation de la sémantique que nous avons décrite précédemment de manière informelle pour chaque attribut associé à un symbole. Nous verrons dans la partie suivante, comment tous ces axiomes sont traduits dans Dedukti.

Enfin, de très nombreux symboles rattachés à la bibliothèque standard de 𝕂 (`hooked-symbol`) sont également ajoutés dans le fichier Kore. Leur règle de traduction est similaire au cas `symbol` : en fonction des attributs qu'ils possèdent, certains axiomes seront générés. Cependant, certains de ces symboles semblent uniquement utiles à des backends particuliers de 𝕂.

7 Traduire Kore dans Dedukti

Cette section s'intéresse à l'outil KaMeLo [4] en cours de développement. Nous présentons une formalisation de la traduction effectuée par cet outil, c'est-à-dire les encodages superficiels réalisés, et formalisés à la figure 10. Une version en 𝕂, en Kore et en Dedukti de IMP, est également disponible avec le code source de KaMeLo.

7.1 Traduction élémentaire

Une très grande majorité des encodages sont assez directs, notamment car 𝕂 et Dedukti partagent un très grand nombre de fonctionnalités, comme les règles de réécriture.

Module et import. Tous les modules sont fusionnés dans un unique fichier, en ajoutant le prélude de Dedukti après la traduction du module `BASIC-K`, puisque les déclarations disponibles dans notre prélude dépendent de la sorte `SortK`.

Sortes et symboles. La très grande majorité des sortes et des symboles traduits dans le fichier Kore découlent des déclarations `syntax` et `configuration` présentes dans 𝕂. Nous constatons que la traduction est très simple : toutes les sortes deviennent des symboles de type `SortK`, sauf la sorte `SortK` elle-même, qui a le type `TYPE`. Pour les symboles, la seule différence est que la signature est currifiée. Les sortes et symboles rattachés à la bibliothèque standard de 𝕂 doivent également être définis dans le prélude de Dedukti : les traductions sont similaires à celles présentées à la figure 10.

Traduire les axiomes. Nous notons X-axiome, un axiome de la Matching Logic commençant par l'opérateur X. Un $\exists$-axiome possède soit l'attribut `subsort` (*axiome de sous-typage*), soit l'attribut `functional` (*axiome de fonctionnalité*). Un $=$-axiome est un axiome équationnel ayant l'attribut `assoc`, `comm`, `unit` ou `idem`. Les $\vee$-axiome et $\bot$-axiome possèdent l'attribut `constructor` et sont des axiomes d'exhaustivité des constructeurs. Un $\neg$-axiome possède l'attribut `constructor` et est un axiome de non-recouvrement des constructeurs, tandis qu'un $\rightarrow$-axiome possédant l'attribut `constructor` est un axiome d'injectivité des constructeurs. Les autres $\rightarrow$-axiomes peuvent avoir l'attribut `initializer`, `projection`, `predicate` ou aucun attribut. Un $\hookrightarrow$-axiome correspond à une règle de réécriture.
Seuls les $\hookrightarrow$-axiome et $\rightarrow$-axiome, n'ayant pas l'attribut `constructor`, sont traduits. Ils ont été générés par une règle de réécriture écrite dans $\mathbb{K}$, et seront donc traduits par une ou des règles de réécriture en Dedukti. La traduction $||\ .\ ||_{\texttt{CTRS}}$ est expliquée aux sections 7.2 et 7.3.

Importation

$|| \texttt{import}\ I\ [\,]\ ||_{\texttt{dk}} = \textit{nothing}$

Sorte

$|| \texttt{sort SortK}\{\}\ [\,]\ ||_{\texttt{dk}} = \texttt{constant symbol}\ \widehat{\texttt{SortK}} : \texttt{TYPE}$

$|| \texttt{sort}\ S\{\}\ [\,]\ ||_{\texttt{dk}} = \texttt{constant symbol}\ \widehat{S} : \widehat{\texttt{SortK}}$ Si $S \neq \texttt{SortK}$.

$|| \texttt{hooked-sort}\ S\{\}\ [\,]\ ||_{\texttt{dk}} = \textit{nothing}$ Si S est définie dans le prélude de Dedukti.

Symbole

$|| \texttt{symbol}\ C\{\}(T_1, ..., T_n) : T_e\ [\ Attr\ ]\ ||_{\texttt{dk}} =$
$\quad \texttt{symbol}\ \widehat{C} : \widehat{T_1} \rightarrow ... \rightarrow \widehat{T_n} \rightarrow \widehat{T_e}$

$|| \texttt{hooked-symbol}\ C\{\}\ [\,]\ ||_{\texttt{dk}} = \textit{nothing}$ Si C est définie dans le prélude de Dedukti.

Axiome

$|| \texttt{alias}\ A\{\}(T_1, ..., T_n) : T_e\ \texttt{where}\ A\{\}(N_1 : T_1, ..., N_n : T_n)\ :=\ Cond\ \wedge\ LHS\ [\,]$
$\texttt{axiom}\ A\{\}(N_1 : T_1, ..., N_n : T_n) \hookrightarrow \top\ \wedge\ RHS\ [Attr]\ ||_{\texttt{dk}} =$
$\quad \begin{cases} \texttt{rule}\ \widehat{LHS}^r \hookrightarrow \widehat{RHS}^r & \text{si } Cond = \top. \\ ||\ (\widehat{LHS}^r,\ \widehat{RHS}^r,\ \widehat{Cond}^r)\ ||_{\texttt{CTRS}} & \text{sinon.} \end{cases}$

$|| \texttt{axiom}\ Ax\ [\ x\ ]\ ||_{\texttt{dk}} = \textit{nothing}$ Si $x \in \{\texttt{subsort}, \texttt{functional}, \texttt{constructor}, \texttt{assoc}, \texttt{comm}, \texttt{unit}, \texttt{idem}\}$.

$|| \texttt{axiom}\ Cond\ \wedge\ Data\ \rightarrow$ Où $Attr$ peut être vide ou contenir les attributs `projection`, `predicate` ou `initializer`.
$\quad (LHS\ =\ (RHS\ \wedge\ \top))\ [Attr]\ ||_{\texttt{dk}} =$
$\quad \begin{cases} \texttt{rule}\ \widehat{LHS}^{Data} \hookrightarrow \widehat{RHS}^{Data} & \text{si } Cond = \top. \\ ||\ (\widehat{LHS}^{Data},\ \widehat{RHS}^{Data},\ \widehat{Cond}^r)\ ||_{\texttt{CTRS}} & \text{sinon.} \end{cases}$

où $\widehat{S}$ transforme l'identificateur S de Kore, en un identificateur de Dedukti.
$\widehat{S}^r$ transforme le pattern S en un terme de Dedukti.
$\widehat{S}^{Data}$ transforme le pattern S en un terme de Dedukti, en utilisant les associations entre variables et patterns décrites par $Data$.

Figure 10 – Traduction de Kore vers Dedukti via KaMeLo

7.2 Traduction des règles de réécriture conditionnelles

Dans cette section, nous nous intéressons à la traduction des règles de réécriture conditionnelles. Comme Dedukti ne propose pas la possibilité de définir des règles de réécriture conditionnelles, il est nécessaire de trouver un encodage d'un système de réécriture conditionnelle ($CTRS$) vers un système de réécriture non conditionnelle (TRS).

7.2.1 Encodage sur un exemple

Considérons le système suivant :
(1) `rule` $max\ X\ Y$ `=>` Y `requires` X `<Int` Y
(2) `rule` $max\ X\ Y$ `=>` X `requires` X `>=Int` Y.
L'encodage permet d'obtenir le système de réécriture suivant dans Dedukti :
(0) `rule` max `$x` `$y` $\hookrightarrow$ $\flat max$ `$x` `$y` $\flat$ $\flat$
(1′) `rule` $\flat max$ `$x` `$y` $\flat$ `$c` $\hookrightarrow$ $\flat max$ `$x` `$y` (`$x` $<$ `$y`) `$c`
(1″) `rule` $\flat max$ `$x` `$y` $true$ `$c` $\hookrightarrow$ `$y`
(2′) `rule` $\flat max$ `$x` `$y` `$c` $\flat$ $\hookrightarrow$ $\flat max$ `$x` `$y` `$c` (`$x` $\geq$ `$y`)
(2″) `rule` $\flat max$ `$x` `$y` `$c` $true$ $\hookrightarrow$ `$x`.
Avec le système obtenu, $max\ 5\ 3$ se calcule ainsi : $max\ 5\ 3 \hookrightarrow_0 \flat max\ 5\ 3\ \flat\ \flat \hookrightarrow_{1'} \flat max\ 5\ 3\ (5 < 3)\ \flat \hookrightarrow^* \flat max\ 5\ 3\ false\ \flat \hookrightarrow_{2'} \flat max\ 5\ 3\ false\ (5 \geq 3) \hookrightarrow^* \flat max\ 5\ 3\ false\ true \hookrightarrow_{2''} 5$.

L'idée générale de l'encodage, proposé dans cette section et initialement proposée par Viry [30], est d'ajouter, pour un symbole défini avec des règles conditionnelles, autant d'arguments qu'il y a de conditions. La règle (0) réécrit un terme dont le symbole de tête est max, par un terme utilisant la version étendue correspondante d'arité 4, $\flat max$, où tous les arguments booléens valent $\flat$, indiquant que les arguments booléens n'ont pas encore été initialisés par une condition. Les règles (1′) et (2′) initialisent les conditions à calculer, tandis que les règles (1″) et (2″) réduisent la taille du terme puisque une des conditions a été évaluée à $true$.

Contrairement à Viry, nous choisissons d'étendre la signature, comme ici avec le symbole $\flat max$, plutôt que de remplacer chaque symbole de la signature par un symbole équivalent mais avec une arité plus grande, permettant de transporter les calculs des conditions. En effet, cela complique le code et oblige à traduire, après coup, les formes normales obtenues.

De plus, cet encodage a l'avantage de ne pas fixer l'ordre d'évaluation des conditions, mais augmente le temps de calcul, cela en doublant le nombre initial de règles.

Enfin, pour générer les règles précédentes, il faut connaître toutes les conditions qui peuvent s'appliquer, pour un symbole de tête donné. Cependant, de nombreuses règles de réécriture écrites dans 𝕂, c'est-à-dire celles ayant un symbole de constructeur comme symbole de tête et n'ayant pas l'attribut `anywhere`, nécessitent d'inférer la configuration. Cela implique que si nous considérons la définition usuelle d'un symbole de tête, ces règles de réécriture auraient le même symbole de tête. Dans le cadre de cet article, nous considérons donc que le symbole de tête d'une règle de réécriture correspond au symbole de tête de la partie du terme gauche de la règle se trouvant dans la cellule `<k>`, sans considérer les symboles `dotk`, `kseq` et `inj`. Si la règle a un symbole de fonction comme symbole de tête ou possède l'attribut `anywhere`, le symbole de tête correspond au symbole de tête au sens usuel.
Nous notons $head_{<k>}$ la fonction qui renvoie le symbole de tête de la cellule `<k>`, pour une règle donnée, sans considérer les symboles `dotk`, `kseq` et `inj`. Nous notons également $\mathcal{C}_\sigma$, l'ensemble des règles qui partagent le même symbole de tête σ, soit $\mathcal{C}_\sigma = \{\ l \overset{c}{\hookrightarrow} r \mid head_{<k>}\ (l \overset{c}{\hookrightarrow} r) = \sigma\ \}$.

7.2.2 Formalisation de l'encodage

Soit $\mathfrak{R}$ un système de réécriture conditionnelle écrit dans 𝕂. Pour éviter des conflits de nommage, nous supposons également que $\flat$ est un nom de symbole non utilisé, et qu'il n'apparaît en tête d'aucun nom de symbole. Nous présentons la traduction notée $||\ .\ ||_{\texttt{CTRS}}$ précédemment. Celle-ci prend en argument un ensemble $\mathcal{E}_{DK}$ de triplets de termes Dedukti de la forme $(LHS,\ RHS,\ c)$, noté ici $LHS \overset{c}{\hookrightarrow} RHS$, obtenu une fois les traductions $||\ .\ ||_{\texttt{kore}}$ et $||\ .\ ||_{\texttt{dk}}$ appliquées sur $\mathfrak{R}$. Après avoir construit les $\mathcal{C}_\sigma$ à partir de $\mathcal{E}_{DK}$, nous déroulons l'algorithme présenté à la figure 11, pour chaque $\mathcal{C}_\sigma$, où X est le nombre de règles conditionnelles dans $\mathcal{C}_\sigma$.

1. Si $X = 0$, $\mathcal{C}_\sigma$ est inchangé et l'algorithme s'arrête. Sinon, initialiser i à 0 et aller en 2.
2. Générer le LHS le plus général pour un symbole σ donné, noté $mglhs_\sigma$.
3. Générer le symbole étendu $\flat\sigma$ de type $T_1 \to ... \to T_{n-1} \to \flat\texttt{Bool} \to ... \to \flat\texttt{Bool} \to T_n$, avec X argument(s) de type $\flat\texttt{Bool}$, où $\flat\texttt{Bool} = \texttt{Bool} \cup \{\flat\}$, et σ de type $T_1 \to ... \to T_n$.
4. Générer la règle de substitution : $mglhs_\sigma \hookrightarrow mglhs_\sigma[\ update_{same}(\sigma, \flat\sigma, \flat)\ ]_\sigma$
5. Pour chaque règle appartenant à $\mathcal{C}_\sigma$:
   (a) Si la règle est de la forme $l \overset{c}{\hookrightarrow} r \in \mathcal{C}_\sigma$ et n'a pas l'attribut `owise` (avec $c \neq \top$) :
   — Incrémenter i de 1
   — Générer une règle d'initialisation :
   $l\ [\ update_{diff}(\sigma, \flat\sigma, \flat, i, \_)\ ]_\sigma \hookrightarrow l\ [\ update_{diff}(\sigma, \flat\sigma, c, i, \_)\ ]_\sigma$
   — Générer une règle de réduction : $l\ [\ update_{diff}(\sigma, \flat\sigma, \texttt{true}, i, \_)\ ]_\sigma \hookrightarrow r$
   (b) Si la règle est de la forme $l \overset{\top}{\hookrightarrow} r \in \mathcal{C}_\sigma$ et n'a pas l'attribut `owise`, générer la règle de réduction : $l\ [\ update_{same}(\sigma, \flat\sigma, \_)\ ]_\sigma \hookrightarrow r$.
   (c) Si la règle $l \overset{\top}{\hookrightarrow} r \in \mathcal{C}_\sigma$ à l'attribut `owise`, générer la règle de réduction : $l\ [\ update_{same}(\sigma, \flat\sigma, \texttt{false})\ ]_\sigma \hookrightarrow r$.

Où :
* $t_1[t_2]_\sigma$ signifie que nous substituons t_2 au sous-terme ayant pour symbole de tête σ dans t_1,
* $arg_i(t)$ correspond au i-ème argument de t et $arity(t)$ au nombre d'arguments de t,
* $mglhs_\sigma$ désigne la configuration initiale où les cellules les plus profondes deviennent, `<c>` y_i si `<c>` $\neq$ `<k>`, `<k>` (`kseq` (`inj` σ z_1 ... $z_{arity(\sigma)}$) L) sinon, avec y_i et z_i des variables fraîches et, L une K computation.
* $update_{diff}(\sigma, \flat\sigma, s_1, i, s_2) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ $avec$ $x_j = \begin{cases} arg_j(\sigma) & \text{si } 1 \leq j \leq arity(\sigma) \\ s_1 & \text{si } j = arity(\sigma) + i \\ s_2 & \text{sinon} \end{cases}$
* $update_{same}(\sigma, \flat\sigma, s) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ $avec$ $x_j = \begin{cases} arg_j(\sigma) & \text{si } 1 \leq j \leq arity(\sigma) \\ s & \text{sinon} \end{cases}$

Figure 11 – Variante de l'encodage de Viry

7.2.3 Extension de l'encodage à l'attribut `owise`

Une manière plus succincte d'écrire l'exemple précédent est d'utiliser l'attribut `owise` :
`rule` $max\ X\ Y$ `=>` Y `requires` X `<Int` Y
`rule` $max\ X\ Y$ `=>` X [`owise`]
Malheureusement, 𝕂 ne génère pas la condition complémentaire dans le fichier Kore. Pour encoder cet attribut, deux possibilités s'offrent à nous : implémenter un algorithme qui détermine la condition complémentaire ou considérer que toutes les conditions se réduisent nécessairement soit à `true`, soit à `false`. Comme nous ne connaissons pas exactement l'expressivité des conditions pouvant être écrites dans 𝕂, outre qu'elles sont de type `Bool`, nous préférons ajouter l'hypothèse suivante : toute fonction construisant un booléen est une fonction totale.
Sous cette hypothèse, nous pouvons générer la règle présentée en 5.(c) (Figure 11).

De plus, 𝕂 accepte un ensemble de règles non conditionnelles avec au moins une règle ayant l'attribut `owise`. Nous excluons ce cas car nous ne pouvons pas modéliser « Si aucune autre règle ne s'applique », avec une condition booléenne. Cela est équivalent à l'utilisation de l'attribut `priority(`*nb*`)`, que nous ne traitons pas encore actuellement.

7.3 Traduction des stratégies d'évaluation

Comme nous l'avons vu à la section 6.2, il est possible que lors de la traduction de 𝕂 vers Kore, des règles de réécriture conditionnelles soient générées, comme c'est le cas pour les stratégies d'évaluation définies par les attributs `strict` et `seqstrict`. Les règles de réécriture générées par ces attributs nécessitent de traduire les K computations, c'est-à-dire les symboles `.` et $\curvearrowright$, les freezers mais aussi le prédicat `isKResult`. L'encodage de ces différents symboles ne pose donc aucune difficulté puisque les règles présentées à la figure 10 pour les symboles s'appliquent aisément. Cependant, ces règles de réécriture conditionnelles s'inscrivent dans un cas de figure connu où l'encodage de Viry n'est pas confluent, notamment car l'ordre d'application de certaines règles de réécriture modifie le résultat de la condition, ce qui peut bloquer le calcul. En effet, la figure 12 montre, à droite, la traduction dans Dedukti des règles `rule` E_1 `and` E_2 `=>` $E_1 \curvearrowright (\clubsuit^1_{\text{and}}\ E_2)$ `requires` $E_1 \notin$ `KResult` (règle C) et `rule` $E_1 \curvearrowright (\clubsuit^1_{\text{and}}\ E_2)$ `=>` E_1 `and` E_2 `requires` $E_1 \in$ `KResult` (règle H), générées par l'attribut `strict(1)`, et, à gauche, un exemple d'exécution valide mais bloquée [10].

1	`rule` E_1 *and* $E_2 \curvearrowright$ `$c` $\hookrightarrow$ ♭*and* E_1 E_2 ♭ $\curvearrowright$ `$c`	(*true and true*) *and false* $\curvearrowright$.
2	`rule` ♭*and* E_1 E_2 ♭ $\curvearrowright$ `$c` $\hookrightarrow$ ♭*and* E_1 E_2 (*not*(*isKResult* E_1)) $\curvearrowright$ `$c`	$\hookrightarrow_1$ ♭*and* (*true and true*) *false* ♭ $\curvearrowright$.
3	`rule` ♭*and* E_1 E_2 *true* $\curvearrowright$ `$c` $\hookrightarrow$ $E_1 \curvearrowright (\clubsuit^1_{\text{and}}\ E_2) \curvearrowright$ `$c`	$\hookrightarrow_2$ ♭*and* (*true and true*) *false* (*not*(*isKResult* (*true and true*))) $\curvearrowright$.
4	`rule` $E_1 \curvearrowright (\clubsuit^1_{\text{and}}\ E_2) \curvearrowright$ `$c` $\hookrightarrow (♭\clubsuit^1_{\text{and}}\ E_1\ E_2$ ♭) $\curvearrowright$ `$c`	$\hookrightarrow^*$ ♭*and* (*true and true*) *false true* $\curvearrowright$.
5	`rule` $(♭\clubsuit^1_{\text{and}}\ E_1\ E_2$ ♭) $\curvearrowright$ `$c` $\hookrightarrow (♭\clubsuit^1_{\text{and}}\ E_1\ E_2$ (*isKResult* E_1)) $\curvearrowright$ `$c`	$\hookrightarrow_3$ (*true and true*) $\curvearrowright (\clubsuit^1_{\text{and}}$ *false*) $\curvearrowright$.
6	`rule` $(♭\clubsuit^1_{\text{and}}\ E_1\ E_2$ *true*) $\curvearrowright$ `$c` $\hookrightarrow$ E_1 *and* $E_2 \curvearrowright$ `$c`	$\hookrightarrow_4 (♭\clubsuit^1_{\text{and}}$ (*true and true*) *false* ♭) $\curvearrowright$.
7	`rule` *true and* `$b` $\curvearrowright$ `$c` $\hookrightarrow$ `$b` $\curvearrowright$ `$c`	$\hookrightarrow_5 (♭\clubsuit^1_{\text{and}}$ (*true and true*) *false* (*isKResult* (*true and true*))) $\curvearrowright$.
8	`rule` *false and* _ $\curvearrowright$ `$c` $\hookrightarrow$ *false* $\curvearrowright$ `$c`	$\hookrightarrow^*$ $(♭\clubsuit^1_{\text{and}}$ (*true and true*) *false false*) $\curvearrowright$.

Figure 12 – Règles générées avec l'encodage précédent et une exécution bloquée

Intuitivement, ces règles servent à s'assurer que E_1 est bien d'un certain type, afin d'autoriser ou non son évaluation. L'idée de notre nouvel encodage est de spécialiser les termes de gauche des règles, c'est-à-dire d'affiner le pattern-matching afin d'assurer le type souhaité pour E_1. Les axiomes de sous-typage générés lors de la traduction de 𝕂 vers Kore nous permettent de connaitre la valeur booléenne de `isKResult` E_1 afin d'affiner correctement le pattern-matching. Nous n'appliquons donc plus notre encodage basé sur Viry dans ce cas de figure, puisque les conditions initiales deviennent inutiles.

Nous illustrons ce nouvel encodage sur l'exemple précédent.

La règle H est compilée en une règle de réécriture non conditionnelle en spécialisant le terme gauche en un terme correspondant à un booléen, soit `rule` $(inj\{Bool\}\{KItem\}$ `$E1`$) \curvearrowright (\clubsuit^1_{\text{and}}$ `$E2`$) \curvearrowright$ `$c` $\hookrightarrow (inj\{Bool\}\{BExp\}$ `$E1`$)$ *and* `$E2` $\curvearrowright$ `$c`. Ainsi la règle H ne pourra être utilisée que si le terme est bien une expression booléenne complètement calculée. Pour la règle qui demande l'évaluation de la première expression, la règle C, nous appliquons le même principe et compilons cette règle non conditionnelle en autant de règles conditionnelles qu'il y a de cas possibles dans la sorte **BExp**, le cas de la constante excepté, soit :

1. `rule` (`$X1` *and* `$X2`) *and* `$E2` $\curvearrowright$ `$c` $\hookrightarrow$ (`$X1` *and* `$X2`) $\curvearrowright (\clubsuit^1_{\text{and}}$ `$E2`$) \curvearrowright$ `$c`
2. `rule` (`$X1` $<$ `$X2`) *and* `$E2` $\curvearrowright$ `$c` $\hookrightarrow$ (`$X1` $<$ `$X2`) $\curvearrowright (\clubsuit^1_{\text{and}}$ `$E2`$) \curvearrowright$ `$c`
3. `rule` (*not* `$X1`) *and* `$E2` $\curvearrowright$ `$c` $\hookrightarrow$ (*not* `$X1`) $\curvearrowright (\clubsuit^1_{\text{and}}$ `$E2`$) \curvearrowright$ `$c`

Nous ne donnons pas de formalisation de cette traduction, faute de place, mais l'exemple détaillé ci-dessus s'adapte aux autres cas présentés à la figure 9, et la généralisation est aisée.

10. Il est également possible d'obtenir *false*.

8 Conclusion

Dans cet article, nous nous sommes principalement intéressés à la traduction dans Dedukti de sémantiques écrites en 𝕂, afin de pouvoir exécuter dans Dedukti des programmes écrits dans le langage décrit par la sémantique. Nous nous sommes appuyés sur le format Kore, sans chercher à le certifier. Nous supposons donc que le fichier Kore produit depuis une sémantique 𝕂 est correcte. Ce travail a nécessité la compréhension de la traduction d'une sémantique 𝕂 dans une théorie de la Matching Logic, nommée Kore, mais également de pouvoir traduire des règles conditionnelles en règles non conditionnelles. Dans l'état actuel, le traducteur KaMeLo traduit la syntaxe du langage, les configurations et les symboles, certains attributs et les règles de réécriture.

Cependant, 𝕂 autorise une forme restreinte de réécriture modulo ACUI, à l'aide des attributs `assoc`, `comm`, `unit` et `idem`. Dans la bibliothèque standard de 𝕂, de très rares symboles ont l'un de ces attributs, comme les dictionnaires (`Map`) et les ensembles (`Set`). A l'heure actuelle, afin de pouvoir exécuter un programme écrit dans le langage de IMP, nous avons utilisé une implantation plus classique des dictionnaires qui n'utilise pas cette fonctionnalité, et ainsi modifié très légèrement la sémantique initiale [11], comme le montre la figure 13.

Implémentation des dictionnaires	Modification de la sémantique	n°
`syntax` **Dico** ::= "`.Dico`" | "`(`" `KItem` "`|-->`" `KItem` "`)`" "`,`" **Dico**		
`syntax` **KItem** ::= "`lookup`" `KItem` **Dico** `rule lookup` x ((y `|-->` _) , d) `=>` `lookup` x d `requires` $x \neq y$ `rule lookup` x ((y `|-->` i) , _) `=>` i `requires` $x = y$	`rule` $\langle\langle\ x \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle$ `lookup` x $d \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$	1'
`syntax` **Dico** ::= "`update`" `KItem` `KItem` **Dico** `rule update` x v ((y `|-->` w) , d) `=>` ((y `|-->` w) , (`update` x v d)) `requires` $x \neq y$ `rule update` x v ((y `|-->` _) , d) `=>` ((y `|-->` v) , d) `requires` $x = y$	`rule` $\langle\langle\ x = i; \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle\ s\ \rangle_k\ \langle$ `update` x i $d\ \rangle_{env}\rangle$	10'
`syntax` **Bool** ::= `KItem` "`in_dico`" "`(`" **Dico**"`)`" `rule` x `in_dico` ((y `|-->` _) , d) `=>` x `in_dico`(d) `requires` $x \neq y$ `rule` x `in_dico` ((y `|-->` _) , _) `=>` *true* `requires` $x = y$ `rule` _ `in_dico` (`.Dico`) `=>` *false*	`rule` $\langle\langle$ `var` $x, xl;\ s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle$ `var` $xl;\ s\ \rangle_k\ \langle$ ((x `|-->` 0) , d) $\rangle_{env}\rangle$ `requires` $\neg_{Bool}$ (x `in_dico`(d))	16'

Figure 13 – Nouvelle sémantique de notre langage IMP

Notre outil ne peut donc pas traiter actuellement les sémantiques faisant usage de la réécriture modulo ACUI. Nous avons également supposé que les variables du membre droit d'une règle de réécriture 𝕂 apparaissent nécessairement dans le membre gauche, ce qui est une restriction très commune en théorie de la réécriture. Cependant cette hypothèse est invalidée par la sémantique 𝕂 du langage Michelson [8]. La perspective la plus immédiate de ce premier travail consiste à étendre l'outil KaMeLo de manière à prendre en considération les attributs tels que `priority(nb)` et à mener une expérimentation sur une collection de sémantiques écrites avec 𝕂. Nous comptons également étendre notre transformation d'un CTRS vers un TRS afin de pouvoir garantir la correction et la complétude de celle-ci, en plus de la préservation de la terminaison et de la confluence. Un autre travail futur concerne la réécriture modulo ACUI que nous contournons actuellement. Une piste ici consiste à mettre en place un encodage *ad hoc* de manière à pouvoir utiliser les dictionnaires et les ensembles tels qu'ils sont implantés dans la bibliothèque de 𝕂.

Remerciements : Nous remercions vivement l'équipe 𝕂, notamment Xiaohong Chen, Everett Hildenbrandt, Zhengyao Lin et Dorel Lucanu, pour les réponses rapides à nos multiples questions.

11. Comme les chaînes de caractères ne sont pas natives dans Dedukti, nous considérons qu'une variable est de la forme `var(nb)`. L'égalité sur les identifiants des variables revient donc à une égalité sur les entiers.

Bibliographie

[1] GitHub avec la formalisation de 𝕂 en 𝕂. https://github.com/kframework/k-in-k.

[2] GitHub de Dedukti. https://github.com/Deducteam/Dedukti.

[3] GitHub de Dedukti v3. https://github.com/Deducteam/lambdapi.

[4] GitLab de KaMeLo. https://gitlab.com/semantiko/kamelo.

[5] Site web de l'école d'été ISR 2021. https://dalila.sip.ucm.es/isr2021/.

[6] Site web de 𝕂. https://kframework.org/.

[7] Site web de Sail. https://www.cl.cam.ac.uk/~pes20/sail/.

[8] Sémantique 𝕂 de Michelson. https://github.com/runtimeverification/michelson-semantics.

[9] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. 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.

[10] F. Blanqui, G. Dowek, E. Grienenberger, G. Hondet, and F. Thiré. Some Axioms for Mathematics. In N. Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20 :1–20 :19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] D. Bogdănaş and G. 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.

[12] P. Borras, D. Clement, T. Despeyroux, J. Bertot, G. Kahn, B. Lang, and V. Pascual. Centaur : The system. 24 :14–24, 03 1989.

[13] X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM, July 2021.

[14] X. Chen, D. Lucanu, and G. Roşu. Matching Logic Explained. Technical Report http ://hdl.handle.net/2142/107794, University of Illinois at Urbana-Champaign and Alexandru Ioan Cuza University, July 2020.

[15] X. Chen and G. Roşu. Applicative Matching Logic : Semantics of K. Technical Report http ://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign, July 2019.

[16] X. Chen and G. Rosu. Matching mu-Logic : Foundation of K Framework (Invited Paper). page 4 pages, 2019. Artwork Size : 4 pages Medium : application/pdf Publisher : Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number : 1.0.

[17] D. Cousineau and G. Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.

[18] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*, pages 243–320, 1990.

[19] G. Dowek. Interacting Safely with an Unsafe Environment. *CoRR*, abs/2107.07662, 2021.

[20] C. Hathhorn, C. Ellison, and G. 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.

[21] L. Li and E. L. Gunter. IsaK-static : A complete static semantics of K. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings*, pages 196–215. Springer-Verlag Berlin Heidelberg, 2018.

[22] L. Li and E. L. Gunter. A Complete Semantics of 𝕂 and Its Translation to Isabelle. In A. Cerone and P. C. Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021*, pages 152–171, Cham, 2021. Springer International Publishing.

[23] D. Mulligan, S. Owens, K. Gray, T. Ridge, and P. Sewell. Lem : Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, 49, 08 2014.

[24] D. Park, A. Ştefănescu, and G. 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.

[25] G. Rosu. K - A Semantic Framework for Programming Languages and Formal Analysis Tools. In D. Peled and A. Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017. From the lecture notes presented at Marktoberdorf'16.

[26] G. Roșu and T. F. Șerbănuță. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6) :397–434, Aug. 2010.

[27] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott : Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1) :71–122, 2010.

[28] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. 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, Oct. 2016. ACM.

[29] 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 R. Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[30] P. Viry. Elimination of Conditions. *Journal of Symbolic Computation*, 28(3) :381–401, 1999.