Encodage du langage mathématique d'Event-B dans Lambdapi

Anne Grieu, Jean-Paul Bodeveix

Encodage du langage mathématique d'Event-B dans Lambdapi

Anne Grieu[1] et Jean-Paul Bodeveix[1]

[1]IRIT, Université de Toulouse, CNRS, INP, UT3, Toulouse, France , prenom.nom@irit.fr

B, Event-B et TLA sont des méthodes formelles basées sur la théorie des ensembles. Dedukti/Lambdapi est un framework logique basé sur le $\lambda\Pi$-calcul modulo dans lequel de nombreuses théories peuvent être exprimées. Dans le projet ANR ICSPA, Lambdapi a été choisi pour échanger les méthodes formelles basées sur la théorie des ensemble B, Event-B et TLA.

Nous allons présenter plus particulièrement certains choix et questionnements concernant l'encodage du langage mathématique d'Event-B et des preuves produites par la plate-forme Rodin, en nous appuyant sur la construction du langage mathématique d'Event-B et sa représentation en Lambdapi. Nous présenterons aussi la représentation des preuves dans Rodin et les propositions de transcription dans Lambdapi.

1 Contexte

Les méthodes formelles sont utilisées pour le développement de logiciels ou de systèmes pour lesquels on a besoin d'un haut niveau de confiance. Elles ont pour but de formaliser le comportement des systèmes et les propriétés attendues de ces comportements, puis de les prouver conformes. Le logiciel de preuve décharge l'utilisateur des preuves automatisables et l'assiste dans les autres. Les théories développées doivent être suffisamment expressives pour énoncer les problèmes que l'on rencontre et permettre leur démonstration automatique autant que faire se peut.

Le langage des prédicats du premier ordre égalitaire, associé à la théorie des ensembles, offre une théorie expressive dans laquelle de nombreux théorèmes sont démontrables automatiquement. Il est la base de plusieurs formalismes comme la méthode B[Abr96], Event-B[Abr10], TLA+[Lam02]. Les résultats présentés ici ont été développés dans le cadre du projet ANR ICSPA [1] qui a pour objectif de renforcer la confiance dans les preuves obtenues avec ces trois formalismes basés sur la théorie des ensembles et d'avoir un cadre de partage pour leurs preuves, qui permettrait une inter-opérabilité de leurs outils respectifs, Atelier B[CLE24], Rodin[ABH+10] et TLAPS[CDL+12]. Le cadre choisi pour ce projet est Dedukti [2]/Lambdapi [CD07, ABC+16], un framework logique basé sur la théorie des types, adapté pour exprimer d'autres théories logiques.

Le projet B-Ware [DDMM14] pour Atelier B et ses récents développements [SHG24] ou la méthode de reconstruction de preuves de TLAPS [Ale23], partagent les mêmes objectifs de vérification des preuves et d'inter-opérabilité à travers Dedukti/Lambdapi mais ont des approches différentes. Dans le projet B-Ware, les obligations de preuve sont traduites vers la

1. https ://anr.fr/Projet-ANR-21-CE25-0015
2. `https://deducteam.github.io/`

plate-forme Why3[BFMP12][why] qui appelle des outils de preuve automatique. Ces preuves sont ensuite vérifiées après traduction de Why3 vers Dedukti. La preuve est faite par des outils externes, si ceux-ci y parviennent, mais les techniques développées par Atelier B pour résoudre ces obligations de preuve ne sont pas réutilisées.

Avec Event-B, notre ambition est de traduire, non seulement les obligations de preuve, mais aussi leurs preuves réalisées à l'aide de la plate-forme Rodin et nous avons pour objectif de pouvoir intégrer aussi les preuves provenant des prouveurs internes et externes de Rodin. Ce travail en est encore à ses débuts. Notre premier objectif est d'être capable de vérifier dans le framework Lambdapi les preuves faite à l'aide de Rodin, en se restreignant aux preuves réalisées sans appels aux prouveurs externes ou internes dans un premier temps, en utilisant uniquement les règles de déduction du langage mathématique d'Event-B.

2 Lambdapi

Lambdapi[CD07] est un assistant de preuve basé sur le $\lambda\Pi$-calcul modulo théorie, un λ-calcul avec des types dépendants et la possibilité d'ajouter des règles de réécriture. Lambdapi n'a pas de logique prédéfinie et permet d'encoder facilement diverses logiques et de vérifier les preuves faites en utilisant ces logiques. Des bibliothèques de logiques fréquemment utilisées [3] sont disponibles, en particulier la logique intuitionniste des prédicats du premier ordre, qui va nous intéresser par la suite.

2.1 Éléments de syntaxe du $\lambda\Pi$-calcul modulo

Nous présentons quelques éléments de syntaxe de Lambdapi. La syntaxe des termes est donnée par :

`t` ::=	`V`	variable
	`TYPE`	sorte des types
	Π `(V:t), t`	produit dépendant
	λ `(V:t), t`	abstraction
	`t t`	application

`A→B` est la notation simplifiée pour Π`(x:A),B` quand x n'a pas d'occurrence libre dans B.

Les extraits de code qui suivent servent à présenter quelques éléments de Lambdapi. Ils sont extraits de la modélisation de la logique intuitionniste du premier ordre proposée par la bibliothèque appelée *lambdapi-stdlib* [4], utilisée dans notre proposition de traduction du langage mathématique d'Event-B vers Lambdapi.

La commande `symbol` permet de déclarer un nouveau symbole de différentes manières, en précisant son type. On peut ainsi déclarer les signatures des opérateurs de la logique :

```
constant symbol Prop : TYPE; // type des formules propositionnelles
constant symbol ⊤ : Prop;    constant symbol ⊥ : Prop;
constant symbol ∧ : Prop → Prop → Prop; notation ∧ infix right 7;
```

La fonction `injective symbol π : Prop → TYPE;` est un plongement des propositions vers les types Lambdapi. Elle associe à chaque formule le type de ses preuves. Cette fonction assure la mise en œuvre de la correspondance de Curry-Howard[How80], transformant la vérification de la preuve d'une formule en une vérification de type dans Lambdapi. Ceci est illustré par les règles d'introduction et d'élimination de la conjonction :

```
constant symbol ∧ᵢ p q : π p → π q → π (p ∧ q);
symbol ∧ₑ1 p q : π (p ∧ q) → π p;   symbol ∧ₑ2 p q : π (p ∧ q) → π q;
```

3. https://github.com/Deducteam/lambdapi-logics
4. https://github.com/Deducteam/lambdapi-stdlib

La preuve de l'implication ⇒ est définie en identifiant, avec une règle Lambdapi [5], une preuve de l'implication entre p et q et une fonction associant une preuve de q à une preuve de p. La commande `symbol identifiant ≔ terme` permet de définir la négation à partir de symboles déclarés auparavant :

```
constant symbol ⇒ : Prop → Prop → Prop; notation ⇒ infix right 5;
rule π ($p ⇒ $q) ↪ π $p → π $q; // la preuve d'une implication est une fonction
symbol ¬ p ≔ p ⇒ ⊥; // Négation introduite via une définition
```

La commande `rule` autorise l'utilisateur à donner ses règles de réécriture, modulo lesquelles le système va raisonner. La vérification de la confluence et de la terminaison du jeu de règles est laissée au soin de l'utilisateur, qui dispose d'appels à des outils externes. De plus, Lambdapi vérifie que la confluence locale est bien préservée et signale les paires critiques non-joignables.

2.2 Approches pour le plongement d'Event-B dans Lambdapi

Pour exprimer les preuves construites par la plate-forme Rodin, il faut déjà choisir une représentation du langage mathématique d'Event-B dans Lambdapi, c'est-à-dire du calcul des prédicats classique avec égalité, étendu avec des axiomes de la théorie des ensembles typée.

Un premier choix possible est un plongement profond d'Event-B dans Lambdapi, en encodant directement les différents connecteurs d'Event-B et les axiomes qui les régissent au-dessus du noyau de Lambdapi.

On peut aussi réaliser un traitement différent pour la logique et la théorie des ensembles. Pour la partie logique, on effectue un plongement superficiel dans les opérateurs introduits par la bibliothèque intuitionniste *lambdapi-stdlib*, qui est elle-même définie par un plongement profond dans Lambdapi. Au-dessus de cette base logique, on peut alors suivre une approche axiomatique pour définir la théorie des ensembles d'Event-B. C'est cette seconde approche qui a été choisie, dans l'optique de faciliter les échanges de preuves entre plusieurs systèmes basés sur le théorie des ensembles. La traduction proposée actuellement assimile les opérateurs logiques d'Event-B à ceux de la bibliothèque *lambdapi-stdlib* évoquée ci-dessus. Cette bibliothèque définit une logique des prédicats du premier ordre *intuitionniste*. Les opérateurs $\wedge, \vee, \bot, \top, \Leftrightarrow, ..$ sont définis en partant de leur signature, puis en ajoutant les axiomes d'élimination et d'introduction associés, comme on a pu le voir sur quelques exemples dans la section précédente. En ajoutant l'axiome du tiers-exclu : `symbol classic (P : Prop): π (P ∨ ¬ P);` et en identifiant les connecteurs logiques d'Event-B avec ceux de la bibliothèque standard, on peut démontrer les mêmes théorèmes que ceux qui sont issus de la théorie mathématique d'Event-B. Les jeux d'axiomes et la déclaration des connecteurs sont différents, mais les théories définies sont identiques. Par exemple, l'opérateur $\vee$ est *axiomatisé* dans bibliothèque standard Lambdapi par ses règles d'introduction et d'élimination. Dans Event-B(§3.1), cet opérateur est *défini* à partir de $\neg$ et $\Rightarrow$ et les règles équivalentes aux règles d'introduction et d'élimination sont démontrables. Il faut être vigilant, en choisissant cette approche, de ne pas modifier la théorie.

Une autre possibilité, en restant dans un plongement superficiel de la logique d'Event-B, aurait été de définir des opérateurs classiques au-dessus des opérateurs intuitionnistes définis par la bibliothèque standard du premier ordre, comme cela est proposé dans la théorie U de Lambdapi [6]. Par exemple, on peut définir le ET classique : `symbol ∧ᶜ p q ≔ ¬ ¬ p ∧ ¬ ¬ q;` En adoptant cette approche, on ajouterait une couche de logique dans notre traduction.

Actuellement, dans le projet ICSPA, la volonté est d'utiliser la bibliothèque *lambdapi-stdlib*.

5. Avec la commande `rule` et la flèche ↪ ; $ pour signaler une variable dans la règle
6. `https://github.com/Deducteam/lambdapi-logics/blob/master/U/Classic.lp`

3 Exprimer Event-B dans Lambdapi

Le langage formel d'Event-B permet de modéliser un système par raffinement successifs, et d'en vérifier des propriétés exprimées, notamment par des invariants comportementaux.

La théorie mathématique d'Event-B est basée sur la logique classique des prédicats du premier ordre avec égalité, étendue par une théorie des ensembles typée et l'arithmétique. Nous n'allons pas développer toute la construction de la théorie, qui est détaillée dans [Abr96], [Abr10] et les documents d'accompagnement de Rodin [CM09], seulement montrer des différences avec la représentation de la théorie du premier ordre proposée dans la bibliothèque *lambdapi-stdlib*.

3.1 Langage des prédicats d'Event-B

Pour définir le langage des propositions, Abrial part des opérateurs $\wedge, \Rightarrow$ et $\neg$ ainsi que $\bot$, et des axiomes définissant leur comportement. Ce choix d'axiomes, qui n'est pas l'introduction et l'élimination habituels des opérateurs, permet une automatisation des preuves. Les axiomes et la procédure de décision sont donnés figure 1, les règles étant appliquées en chaînage arrière.

	Antecedents	Consequent
INI	$H \vdash \neg R \Rightarrow \bot$	$H \vdash R$
AXM		$H, P, \neg P \vdash R$
AND1	$H \vdash \neg Q \Rightarrow R$ $H \vdash \neg P \Rightarrow R$	$H \vdash \neg(P \wedge Q) \Rightarrow R$
AND2	$H \vdash P \Rightarrow (Q \Rightarrow R)$	$H \vdash (P \wedge Q) \Rightarrow R$
IMP1	$H \vdash P \Rightarrow (\neg Q \Rightarrow R)$	$H \vdash \neg(P \Rightarrow Q) \Rightarrow R$
IMP2	$H \vdash Q \Rightarrow R$ $H \vdash \neg P \Rightarrow R$	$H \vdash (P \Rightarrow Q) \Rightarrow R$
NEG	$H \vdash P \Rightarrow R$	$H \vdash \neg\neg P \Rightarrow R$
DED	$H, P \vdash R$	$H \vdash P \Rightarrow R$

Règles (ordonnées) :	Procédure :
AXM, IMP1, IMP2, AND1, AND2, NEG	INI ; (Règles* ;DED)*

Figure 1. Base du langage des propositions d'Event-B

Les opérateurs dérivés $\vee, \Leftrightarrow$ et $\bot$ sont ensuite dérivés des constructeurs de base [7] :

$$P \vee Q \mathrel{\widehat{=}} \neg P \Rightarrow Q,\ P \Leftrightarrow Q \mathrel{\widehat{=}} (P \Rightarrow Q) \wedge (Q \Rightarrow P) \text{ et } \top \mathrel{\widehat{=}} \neg\bot$$

Ensuite, Abrial définit le langage des prédicats du premier ordre, en introduisant les catégories syntaxiques des expressions et des variables, le quantificateur $\forall$, la notion de substitution, les notions de variables libres et liées. Le quantificateur existentiel $\exists$ est défini comme opérateur dérivé par : $\exists$ `x.P` $\mathrel{\widehat{=}}$ $\neg\forall x.\neg P$.

Enfin, pour la méthode B et Event-B, la théorie avec un prédicat d'égalité est étendue avec des règles d'inférence axiomatiques équivalentes à l'égalité de Leibniz, ainsi qu'avec la notion de paires ordonnées, représentées avec le symbole $\mapsto$ (notion définie indépendamment de la théorie des ensembles). Ce langage permet de démontrer les théorèmes habituels de la logique des prédicats du premier ordre classique égalitaire.

3.2 La logique des prédicats du premier ordre classique dans Lambdapi

Dans la partie 2.1, la présentation du langage Lambdapi a permis d'introduire quelques éléments de traduction du langage des propositions, tirés de la librairie `Prop.lp` : un

7. On notera les réécritures avec $\mathrel{\widehat{=}}$

plongement des propositions dans les types et définition des connecteurs logiques par leurs règles d'introduction et d'élimination et nous ajoutons l'axiome du tiers-exclu. Pour la logique du premier ordre, nous avons aussi besoin des quantificateurs sur des données typées. Pour cela, *lambdapi-stdlib* introduit le type `Set: TYPE` des types des types de données. La fonction τ : `Set` $\rightarrow$ `TYPE` assure le plongement des objets de type `Set` dans `TYPE`. Les quantificateurs [8] sont des constantes génériques dont la sémantique est définie par une règle :

```
constant symbol ∀ [a] : (τ a → Prop) → Prop;  notation ∀ quantifier;
rule π (∀ f) ↪ ∏ x, π(f x); // la preuve d'un (∀ x) est une fonction sur x
constant symbol ∃ [a] : (τ a → Prop) → Prop;  notation ∃ quantifier;
constant symbol ∃ᵢ [a] p (x:τ a) : π (p x) → π (∃ p);
symbol ∃ₑ [a] p : π (∃ p) → Π q, (Π x:τ a, π (p x) → π q) → π q;
rule ∃ₑ _ (∃ᵢ _ $x $px) _ $f ↪ $f $x $px;
```

Enfin, nous utilisons Eq.lp pour représenter l'égalité de Leibniz.

3.3 La théorie typée des ensembles d'Event-B et sa traduction dans Lambdapi

La méthode B/Event-B est dotée d'une théorie des ensembles typée, le typage permettant d'interdire d'écrire des prédicats tels que : $\exists x.\ (x \in x)$.

Constructeurs Les *ensembles* sont définis à partir du prédicat d'appartenance : $E \in s$, pour E une expression et s un ensemble, et des constructeurs de base : produit cartésien, ensemble des parties et ensemble en compréhension. Les règles suivantes, données pour S et T deux ensembles, P un prédicat, E une expression, définissent le comportement de ses opérateurs vis à vis de l'appartenance et introduisent l'égalité entre ensembles, appelé axiome d'extensionnalité :

$S = T$	$\widehat{=}$	$S \in \mathbb{P}(T) \wedge T \in \mathbb{P}(S)$	
$(E \mapsto F) \in S \times T$	$\widehat{=}$	$E \in S \wedge F \in T$	($\mapsto$ constructeur d'une paire ordonnée)
$E \in \mathbb{P}(S)$	$\widehat{=}$	$\forall x.\ x \in E \Rightarrow x \in S$	si x est libre dans E et dans S
$E \in \{x.\ P \mid F\}$	$\widehat{=}$	$\exists x.\ (P \wedge E = F)$	si x est libre dans E

A partir de ces axiomes, l'inclusion et l'inclusion stricte sont définies : $S \subseteq T \mathrel{\widehat{=}} S \in \mathbb{P}(T)$ et $S \subset T \mathrel{\widehat{=}} S \subseteq T \wedge S \neq T$.

Les opérateurs classiques, union, intersection, différence, l'ensemble vide, générique, et les ensembles définis en extension ne sont pas considérés comme dérivés, mais eux aussi définis de manière axiomatique. Par exemple, pour l'union : $E \in S \cup T \ \widehat{=}\ E \in S \vee E \in T$.

Ensuite, les relations binaires sur $S \times T$, qui servent de base à la construction des fonctions, sont définies par : $r \in (S \leftrightarrow T) \mathrel{\widehat{=}} r \subseteq S \times T$. Sur cette notion, on définit le domaine, l'image, l'inverse, les restrictions d'une relation et enfin les notions de fonction partielle, totale, surjective, injective, bijective.

Typage dans Event-B La notion de typage introduite dans B et Event-B permet d'éviter des paradoxes comme celui de Russel lors de la définition d'ensembles en compréhension. Le détail de la syntaxe abstraite et des règles de typage utilisées dans Rodin pour Event-B est disponible dans [ABH+10]. Un type d'Event-B est défini par la grammaire suivante :

`T` $:=$	`BOOL` | $\mathbb{Z}$	types prédéfinis : booléens et entiers
	| `S`	ensembles S définis par l'utilisateur
	| $\mathbb{P}$ `T`	ensemble des parties d'un type
	| `T` $\times$ `T`	produit cartésien de types

8. Le modificateur `quantifier` devant `symbol f` permet d'écrire `` `f x, t `` à la place de `f (λ x, t)`.

Ce que nous avons encodé dans Lambdapi par :

```
injective symbol σℙ: Set → Set; // type de l'ensemble des parties d'un type
injective symbol σ×: Set → Set → Set; notation σ× infix left 24; // prd cartésien
constant symbol σBOOL: Set; constant symbol σℤ: Set;
constant symbol σS: Set; // type associé à un ensemble utilisateur S
```

Dans Event-B, on doit vérifier le typage de toute formule. A partir des règles de typage, du contexte, qui contient des déclarations d'ensembles de base et de constantes typées, et de la syntaxe des formules, Rodin infère statiquement le type des différents objets. L'utilisateur peut, dans la majorité des cas se passer de préciser explicitement le type, ce qui est différent du B classique [9]. Chaque étape de vérification de type permet de vérifier la cohérence de type de la formule considérée, expression ou prédicat et enrichit le contexte de nouvelles informations de type.

Constructeurs et opérateurs de la théorie des ensembles Une fois l'encodage des types fixé, on peut définir les opérateurs ensemblistes à partir de l'opérateur générique d'appartenance : `symbol ∈ [T:Set] : τ T → τ (σℙ T) → Prop;`. Par la suite, on déclare la signature des différents objets de la théorie des ensembles d'Event-B avec la commande `symbol` et leur comportement vis-à-vis de ∈ avec la commande `rule`, comme on peut le voir sur l'exemple suivant :

```
injective symbol ↦ [T1 T2:Set] (x:τ T1) (y:τ T2) : τ (T1 σ× T2); // couple
symbol prj1[a b]: τ (a σ× b) → τ a; symbol prj2[a b]: τ (a σ× b) → τ b;
symbol × [T1 T2:Set]: τ (σℙ T1) → τ (σℙ T2) → τ (σℙ (T1 σ× T2));// prd cartésien
rule $x ∈ $A × $B ↪ prj1 $x ∈ $A ∧ prj2 $x ∈ $B;
rule $e ∈ ℙ $S ↪ $e ⊆ $S; // ensemble des parties
constant symbol ∩ [T:Set]: τ (σℙ T) → τ (σℙ T) → τ (σℙ T);  // intersection
notation ∩ infix left 23; rule $x ∈ $s1 ∩ $s2 ↪ $x ∈ $s1 ∧ $x ∈ $s2;
```

Ensemble maximal Un ensemble maximal est un ensemble qui contient tous les éléments d'un type donné. L'appartenance à un tel ensemble se réduit à ⊤. Cette propriété est exploitée implicitement par la tactique Rodin appelée *Typerewrites*. Afin de l'automatiser dans la traduction Lambdapi, nous avons introduit une constante générique `BIG` [10] et les réécritures suivantes :

```
constant symbol BIG [T:Set]: τ (σℙ T);
rule $x ∈ BIG ↪ ⊤; // BIG contient tous les éléments de son type implicite
rule ℙ BIG ↪ BIG; // l'ensemble des parties de BIG est aussi un ensemble maximal
rule BIG × BIG ↪ BIG; // le prd cartésien de deux ens. maximaux est aussi maximal
```

Les ensembles `BIG` sont paramétrés par le type de leurs éléments. L'utilisation de cette constante et ses règles ne suffit pas à résoudre entièrement la difficulté. En effet, le jeu de règles introduit n'est plus confluent, il y a des paires critiques non-joignables, comme on peut le voir dans l'exemple ci-après, montrant deux manières de réécrire $x \in \mathbb{P}(BIG)$. BIG_t est le BIG paramétré par le type implicite de x.

A partir des règles de définition de `BIG`, on a : $x \in \mathbb{P}(BIG_t) \hookrightarrow x \in BIG_t \hookrightarrow \top$

Et à partir de la définition de l'ensemble des parties, puis de l'inclusion, on a :

$$x \in \mathbb{P}(BIG_t) \hookrightarrow x \subseteq BIG_{P(t)} \hookrightarrow \forall u.\ u \in x \Rightarrow u \in BIG_{P(t)} \hookrightarrow \forall u.\ u \in x \Rightarrow \top$$

➥ Il faudrait pouvoir identifier $\top$ et $\forall u.\ u \in x \Rightarrow \top$, mais la bibliothèque sur laquelle notre traduction repose ne propose pas de telles réécritures.

La solution retenue est d'ajouter le modifieur `sequential` dans la déclaration de l'inclusion, pour imposer une priorité sur les règles. Ainsi l'extensionnalité de l'inclusion n'est appliquée que si son deuxième argument n'est pas `BIG`.

9. Dans Event-B, on peut écrire le prédicat : $\forall x.\ x * x \geq 0$, Rodin inférera que le type de x est $\mathbb{Z}$.
10. correspond à `Full_set` dans Coq et UNIV dans HOL

```
sequential symbol ⊆ [T] (s1:τℙ T) (s2: τℙ T): Prop;
rule $s ⊆ BIG ↪ ⊤
with $s1 ⊆ $s2 ↪ `∀ x, x ∈ $s1 ⇒ x ∈ $s2;
```

À l'aide de la constante `BIG`, on définit les ensembles des booléens et des entiers, ainsi que les ensembles définis par l'utilisateur :

```
symbol ℤ: τ (σℙ σℤ) ≔ BIG;
symbol S: τ (σℙ σS) ≔ BIG;  // par ensemble porteur déclaré par l'utilisateur
```

Opérateurs relationnels et fonctionnels La méthode suivie pour définir ces opérateurs est similaire à celle développée dans le paragraphe précédent. Les opérateurs sont définis les uns par rapport aux autres, en ajoutant des propriétés aux opérateurs précédemment définis, une fois les signatures déclarées. Tout est relié à la définition de la relation, avec $E_1 \leftrightarrow E_2$ défini par $\mathbb{P}(E_1 \times E_2)$.

```
symbol rel (T1 T2: Set) ≔ τ (σℙ (T1 σ× T2));
symbol ↔ [T1 T2:Set] (A:τ (σℙ T1)) (B: τ (σℙ T2)):
    τ (σℙ (σℙ (T1 σ× T2))) ≔ ℙ (A × B); notation ↔ infix 11;
constant symbol dom [T1 T2:Set]: rel T1 T2 → τ (σℙ T1); notation dom prefix 30;
rule $x ∈ dom($r) ↪ `∃ y, $x ↦ y ∈ $r;
```

Cette construction par ajout de propriétés pour chaque opérateur, est utilisée pour définir les fonctions partielles, partielles surjectives et totales surjectives d'Event-B. Par exemple, une fonction partielle est définie comme une relation avec des contraintes : $f \in A \nrightarrow B \equiv f \in A \leftrightarrow B \wedge \forall x, \forall y1, \forall y2, x \mapsto y1 \in f \wedge x \mapsto y2 \in f \Rightarrow y1 = y2$.

La traduction Lambdapi respecte cette construction :

```
constant symbol ⇸ [T1 T2:Set]: τ (σℙ T1) → τ (σℙ T2) → τ (σℙ (σℙ (T1 σ× T2)));
notation ⇸ infix 11;
rule $f ∈ ($A⇸$B) ↪ ($f ∈ $A ↔ $B) ∧
    (`∀ x,`∀ y1,`∀ y2, x ↦ y1 ∈ $f ∧ x ↦ y2 ∈ $f ⇒ y1=y2);
```

4 Traduire des preuves de Rodin vers Lambdapi

Cette partie présente succinctement les solutions adoptées et les difficultés rencontrées dans la traduction des preuves de Rodin vers Lambdapi. Pour cela, nous présentons une traduction automatique vers Lambdapi d'une preuve du théorème de Cantor [11] réalisée à l'aide de l'environnement Rodin.

4.1 Arbre de preuve Rodin

Une preuve Rodin est une succession d'application de règles de déduction et de simplifications. Partant du but à démontrer, le prouveur [12] applique des tactiques transformant un séquent en un ensemble de séquents. L'utilisateur peut interagir avec Rodin en cliquant sur des éléments actifs auxquels sont associées des règles prédéfinies applicables. Il peut aussi appeler des prouveurs internes ou externes, mais nous ne discuterons pas de ces appels dans cet article. La capture d'écran 2 illustre cette interaction.

La preuve complète est une preuve entièrement guidée par l'utilisateur, qui ne fait aucun appel automatique à des prouveurs internes, qui ne sont pour l'instant pas traités par la traduction.

11. Soit S un ensemble, il n'existe pas de surjection de S sur l'ensemble des parties de S.
12. Le terme de prouveur désignera l'assistant de preuve Rodin aussi bien que les prouveurs automatiques.

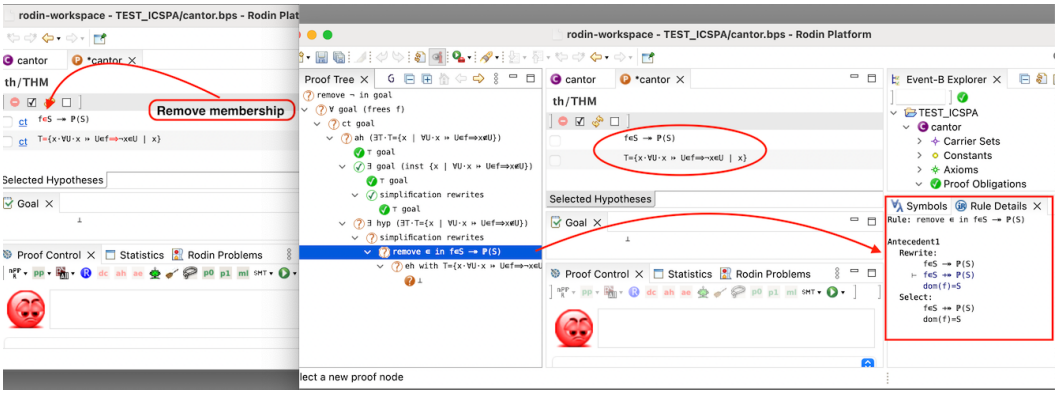

Figure 2. Le prouveur Rodin, avec l'arbre de preuve et le détail des règles utilisées

4.2 Traduire un nœud de l'arbre de preuve

Les actions effectuées par Rodin apparaissent dans l'arbre de preuve. Dans ce document, nous nous intéressons à la manière de traduire certaines de ces actions vers Lambdapi.

La figure 2 montre l'application d'une règle d'inférence, ***remove membership***, suite au clic de l'utilisateur sur le symbole ∈ qui apparaît en rouge. Elle apparaît sous le nom *rule : org.eventb.core.seqprover.rmL1* dans le fichier XML généré par Rodin représentant l'arbre de preuve. Le plug-in Java développé, couplé à Rodin, transforme l'arbre de preuve en script Lambdapi. Les indications fournies par Rodin sont transformées pour constituer une suite de tactiques. Le script généré pour notre exemple de travail, le théorème de Cantor, est proposé en matériel complémentaire.

Tactiques Lambdapi simples. Lambdapi propose un jeu restreint de tactiques pour aider à la résolution des preuves de typage et d'unification. Pour certaines règles de déduction d'Event-B, on pourra appliquer directement une tactique ou une succession de tactiques, comme on peut le voir dans la figure 3.

Règle Rodin	**Lambdapi**	**Règle Rodin**	**Lambdapi**
$\dfrac{}{\Gamma, h : p \vdash p}$ (hyp)	refine h	$\dfrac{\Gamma, h : x_i \in T_i \vdash p}{\Gamma \vdash \forall x_1, \ldots, x_n \cdot p}$ ($\forall$goal)	assume $x_1 \ldots x_n$
$\dfrac{h : p, \Gamma \vdash q}{\Gamma \vdash p \Rightarrow q}$ ($\Rightarrow$ goal)	assume h	$\dfrac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}$ ($\wedge$goal)	apply $\wedge_i \; p \; q$

Figure 3. Traduction de règles simples vers Lambdapi

Application de règles dérivées. Pour de nombreuses règles, nous introduisons des théorèmes dans Lambdapi représentant les règles d'inférence [13] ou de réécriture [14] de Rodin. Par exemple, la règle $DERIV\_IMP\_IMP : \quad P \Rightarrow (Q \Rightarrow R) \;\widehat{=}\; P \wedge Q \Rightarrow R$ qui correspond à l'axiome $AND2$ du langage des propositions, est traduite par un théorème :

```
symbol and_imp [p1 p2 q: Prop]: π (p1 ⇒ p2 ⇒ q) → π ((p1 ∧ p2) ⇒ q) ≔
begin assume p1 p2 q hi hc; apply hi (∧e1 p1 p2 hc) (∧e2 p1 p2 hc) end;
```

13. https://wiki.event-b.org/index.php/Inference_Rules
14. https://wiki.event-b.org/index.php/All_Rewrite_Rules

Opérateurs n-aires de Rodin. Certains opérateurs de Rodin sont n-aires, comme la conjonction, alors que les opérateurs de Lambdapi sont binaires. Le choix actuel a été de transformer ces expressions n-aires en expressions binaires imbriquées à l'aide d'un plug-in Java intégré à Eclipse, sur lequel Rodin est basé.

Par exemple, si le but à prouver est $P_1 \wedge P_2 \wedge P_3$, Rodin génère automatiquement trois sous-buts. Dans Lambdapi, on génère, à l'aide du plug-in Java, une composition de règles d'introduction, dont l'application va construite un nœun n-aire attendant les n sous-buts correspondants : `apply (∧ᵢ (P₁ ∧ (P₂ ∧ P₃)) _ ( ∧ᵢ P₂ P₃ _ _) )`.

Règles de réécriture Lambdapi. Elles sont utilisées pour décrire la sémantique de chaque opérateur ensembliste. Rodin dispose des mêmes réécritures, mais elles sont contrôlées par l'utilisateur et apparaissent explicitement dans l'arbre de preuve. Pour conserver la synchronisation entre la preuve Rodin et la preuve Lambdapi, nous utilisons l'identité pour contrôler l'application des règles au niveau de Lambdapi. Par exemple, la transformation de l'hypothèse $h : x \in A \cup B$ en $x \in A \wedge x \in B$ correspondant à l'application de la règle Rodin *remove* $\in$ *in* h sera traduite en Lambdapi par `have h1 : x ∈ A ∨ x ∈ B {refine h}`.

Transformations avancées. Plusieurs points nécessitent la génération en Java de termes de preuve complexes, à défaut d'un langage de tactiques évolué comme Ltac en Coq.

L'élimination des doublons, automatique dans Rodin, nécessite une composition de théorèmes d'associativité et de commutativité des connecteurs logiques $\wedge$ et $\vee$ dans Lambdapi.

D'autre part, Rodin réalise la réécriture d'un équivalent par un équivalent dans une sous-formule, éventuellement sous des quantificateurs. L'implantation de cette transformation en Lambdapi nécessiterait des mécanismes comme *setoid_rewrite* en Coq[Coe04], à moins de revoir la représentation d'Event-B en Lambdapi pour identifier égalité et équivalence.

Rodin fait aussi appel à de nombreux prouveurs internes ou externes (solveurs SMT). Il faudrait extraire une preuve Lambdapi des traces d'exécution de ces outils ou utiliser un outil comme Zenon-Modulo[DDG+13] sachant produire une trace Lambdapi. Les difficultés sont de différents ordres. Une première difficulté tient à la taille des preuves générées par les outils externes. Une autre, au fait que les traces des outils ne sont pas exprimées dans un format unifié.

5 Conclusion

Après avoir présenté la modélisation en Lambdapi de la logique du premier ordre et de la théorie des ensembles typée d'Event-B, nous avons présenté la traduction de certains types de règles de preuve de Rodin afin de reconstituer une preuve en Lambdapi à partir des arbres de preuve réalisés à l'aide de l'environnement Rodin.

Il reste encore un travail important à accomplir pour obtenir une traduction complète de preuves d'Event-B dans Lambdapi. Traiter toutes les règles de déduction et de simplification représente un travail conséquent, d'une part en raison du nombre de règles à considérer, d'autres part en raison de la complexité de certaines règles d'Event-B.

Enfin, il reste à définir en Lambdapi les notions de machines, d'évènements, les mécanismes de raffinement et les obligations de preuve associées. Notre travail constitue un socle nécessaire pour aborder toutes ces notions.

Références

[ABC+16] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant et 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, mai 2016.

[ABH+10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta et Laurent Voisin : Rodin : an open toolset for modelling and reasoning in Event-B. STTT, 12(6):447–466, 2010.

[Abr96] Jean-Raymond Abrial : The B-book - assigning programs to meanings. Cambridge University Press, 1996.

[Abr10] Jean-Raymond Abrial : Modeling in Event-B - System and Software Engineering. Cambridge University Press, 2010.

[Ale23] Coltellacci Alessio : Reconstruction of TLAPS proofs solved by verit in Lambdapi. In Uwe Glässer, José Creissac Campos, Dominique Méry et Philippe A. Palanque, éditeurs : Rigorous State-Based Methods - 9th International Conference, ABZ 2023, Nancy, France, May 30 - June 2, 2023, Proceedings, volume 14010 de Lecture Notes in Computer Science, pages 375–377. Springer, 2023.

[BFMP12] François Bobot, Jean-Christophe Filliâtre, Claude Marché et Andrei Paskevich : Why3 : Shepherd your herd of provers. Boogie 2011 : First International Workshop on Intermediate Verification Languages, 05 2012.

[CD07] Denis Cousineau et Gilles Dowek : Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, éditeur : Typed Lambda Calculi and Applications, pages 102–117, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts et Hernán Vanzetto : TLA + proofs. In Dimitra Giannakopoulou et Dominique Méry, éditeurs : FM 2012 : Formal Methods, pages 147–154, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CLE24] CLEARSY : Atelier B Tool, 2024. `https://www.atelierb.eu/en/`.

[CM09] Laurent Voisin Christophe Métayer : The Event-B mathematical language, 2009. `https://webarchive.southampton.ac.uk/deployeprints.ecs.soton.ac.uk/11/4/kernel_lang.pdf`.

[Coe04] Claudio Sacerdoti Coen : A semi-reflexive tactic for (sub-)equational reasoning. In Proceedings of the 2004 International Conference on Types for Proofs and Programs, TYPES'04, page 98–114, Berlin, Heidelberg, 2004. Springer-Verlag.

[DDG+13] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand et Olivier Hermant : Zenon Modulo : When Achilles Outruns the Tortoise using Deduction Modulo. In Ken McMillan, Aart Middeldorp et Andrei Voronkov, éditeurs : LPAR - Logic for Programming Artificial Intelligence and Reasoning - 2013, volume 8312 de LNCS, pages 274–290, Stellenbosch, South Africa, décembre 2013. Springer.

[DDMM14] David Delahaye, Catherine Dubois, Claude Marché et David Mentré : The BWare project : Building a proof platform for the automated verification of B proof obligations. In Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477, ABZ 2014, page 290–293, Berlin, Heidelberg, 2014. Springer-Verlag.

[How80] William Alvin Howard : The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger et P. Jonathan, éditeurs : To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus, and Formalism. Academic Press, 1980.

[Lam02] Leslie Lamport : Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.

[SHG24] Claude Stolze, Olivier Hermant et Romain Guillaumé : Towards Formalization and Sharing of Atelier B Proofs with Dedukti. working paper or preprint, janvier 2024.

[why] Why3, a tool for deductive program verification, GNU LGPL 2.1.