From Event-B to Lambdapi (Journées FAC 2024)

Anne Grieu

From Event-B to Lambdapi

Anne Grieu
IRIT, Université de Toulouse, CNRS, INP, UT3, Toulouse, France
anne.grieu@irit.fr

FAC 2024

Abstract

B, Event-B and TLA are development languages based on set theory. Dedukti/Lambdapi is a logical framework based on the $\lambda\Pi$-calculus modulo in which many theories and logics can be expressed. In the context of ICSPA (ANR project), Lambdapi will be used to exchange models and proofs between the set theory-based formal methods B, Event-B and TLA. They will rely on the encoding of the respective set theories in Lambdapi.

We will here focus on Event-B with the Rodin Platform, an Eclipse-based IDE and logical framework (Java-based) for Event-B. In this first work, our aim is to check the proof trees generated by Rodin. For this purpose, we use the Lambdapi framework. We present the translation of Event-B constructs based on the set theory (first order equational classic logic, set based-logic) and some deduction rules. We also discuss how the specific rules of Event-B related to set theory are translated.

1 Context

Deductive formal methods are used to improve confidence in software development, especially for critical systems. The proposal of the ICSPA project (ANR project) is to enable B, Event-B and TLA, all based on set theory, to exchange proofs and models made with their proof systems Atelier B, Rodin and TLAPS.

The framework chosen to express this interoperability is Dedukti/Lambdapi [4, 5], a logical framework based on Martin-Löf's type theory, adapted to express other logical theories. In Lambdapi, we can define objects and types, rewriting rules and reason modulo these rules. Lambdapi has a small core that is easy to verify.

We will here focus on the translation of Event-B mathematical language and proof trees to Lambdapi. We are not concerned by the B-method by itself, i.e., development by refinement.

Event-B [2] is a formal method to specify and model complex systems using the notion of abstract machine notation and Rodin a platform based on Eclipse to manage, edit and prove Event-B models.

We want to formalize the features of the first order classical logic, the typed-set language of Event-B in the $\lambda\Pi$-calculus modulo of Dedukti/Lambdapi. We must be able to define the elements of Event-B such as expressions and predicates and more generally its typed-set theoretical language together with its proof rules.

2 First order classical logic in Lambdapi

2.1 Lambdapi

In this part, we will give some outlines of the syntax of Lambdapi we will encounter in the encoding of Event-B in Lambdapi. We won't go into all details, that can be found in the user manual [1]. We will illustrate the declaration of objects and the definition of rules with examples taken from our formalisation.

[1] `https://lambdapi.readthedocs.io/en/latest/index.html`

2.1.1 Lambdapi terms

The syntax of λΠ terms is given by:

t	::=	V	variable
		$\mid$ `TYPE`	sort for types
		$\mid \Pi\ (V : t),\ t$	dependent product type
		$\mid \lambda\ (V : t),\ t$	abstraction
		$\mid t\ t$	application

The expression $A \to B$ stands for non-dependent product. It is syntactic sugar for $\prod\ (x : A),\ B$ when x has no free occurrence in B.

Examples of Lambdapi terms are given in the following paragraph through the encoding of propositional logic.

2.1.2 Propositions and data type representation in Lambdapi

As it is made in the standard library[2] `Prop.lp`, we define the type of propositions with the `symbol` command. We define as well the type of a proof of a proposition. This will allow us to reduce proof-checking to type-checking. In Lambdapi, TYPE is the sort of types.

```
constant symbol Prop : TYPE; // Propositional logic
injective symbol π : Prop → TYPE; // the type of a Prop proof
```

To express typed-set theory, we define the type Set and how we will interpret sets in TYPE, as in the file `Set.lp` of the standard library.

```
constant symbol Set : TYPE; // Type of set codes for quantifying over sets
injective symbol τ : Set → TYPE; // Interpretation of set codes in TYPE
```

Once these types are fixed, we can continue our encoding.

2.1.3 First order equational logic connectors in Lambdapi

To translate our formalism, we have to define first order logic connectors with their rules. To declare objects with their signature we use the same `symbol` command as to define types. This command can also be used to give a definition, when followed by a term and potentially a type solving or unification goal proof.

The first-order logic of Event-B contains standard operators such as conjunction, disjunction, negation, implication, equivalence, the quantifiers ∀, ∃, two constants ⊤ (truth) and ⊥ (falsity). [1]

In the following examples of the `Prop.lp` file, we can see how these operators are declared with their signature:

```
constant symbol ⊤ : Prop; // True
constant symbol top : π ⊤;
constant symbol ⊥ : Prop; // False
constant symbol ⊥e p : π ⊥ → π p;

// Conjunction
constant symbol ∧ : Prop → Prop → Prop;
```

```
notation ∧ infix left 7;
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;
...
```

It is easy to recognize, in the above excerpt, some rules of the first order logic formalism, like introduction of ∧ and eliminations of ∧ :

[2] https://github.com/Deducteam/lambdapi-stdlib

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}\,(\wedge_i) \qquad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p}\,(\wedge_{e1}) \qquad \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q}\,(\wedge_{e2})$$

We can see a use of the `rule` command in the standard library to declare the implication, by taking benefits of the Curry-Howard correspondence. A proof of an implication is a function taking as parameters a proof of the antecedent and returning a proof of the consequent:

```
constant symbol ⇒ : Prop → Prop → Prop; // Implication
notation ⇒ infix right 5;
rule π ($p ⇒ $q) ↪ π $p → π $q;
```

The negation and the equivalence connectors are introduced with already defined connectors:

```
symbol ¬ p ≔ p ⇒ ⊥; // Negation
symbol ⇔ p q ≔ (p ⇒ q) ∧ (q ⇒ p); // Equivalence
notation ⇔ infix right 5;
```

Let us illustrate the proof of a theorem with the reflexivity of equivalence.

The theorem is introduced with the `symbol` command and appears as a parameter of the π operator. Proving this theorem comes to finding a term of which type is the stated theorem. Some tactics, like `assume`, `apply`, are available to help to solve typing goals or unification goals of Lambdapi.

```
opaque symbol ⇔_refl [p] : π (p ⇔ p) ≔ // ⇔ reflexivity
begin
  assume p;
  apply ∧ᵢ { assume h; apply h } { assume h; apply h }
end;
... // ⇔ symmetry, transitivity
```

We will need universal and existential quantifiers. The definition of the universal quantifier gives an example of use of the dependent product:

```
constant symbol ∀ [a] : (τ a → Prop) → Prop;
notation ∀ quantifier;
rule π (∀ $f) ↪ ∏ x, π ($f x); // Universal quantification

constant symbol ∃ [a] : (τ a → Prop) → Prop;
notation ∃ quantifier; // Existential quantification
constant symbol ∃ᵢ [a] p (x:τ a) : π (p x) → π (∃ p);
symbol ∃ₑ [a] p : π (∃ p) → ∏ q, (∏ x:τ a, π (p x) → π q) → π q;
rule ∃ₑ _ (∃ᵢ _ $x $px) _ $f ↪ $f $x $px;
```

Note: the modifier `quantifier` allows to write `` `f x, t `` instead of `f (λ x, t)`.

Furthermore, we have to declare the law of excluded middle, to be in the context of the classical logic of Event-B:

```
symbol classic (P : Prop): π (P ∨ ¬ P); // Classical
```

2.1.4 Example

As an example, let's consider the formalization in Lambdapi and the proof of the property:

$\forall\ f : T \to T, (\forall\ x\ y, fx = fy) \ \Rightarrow\ \exists\ c,\ \forall\ x,\ fx = c$ for a given non empty type T. After introducing the data type T, the theorem is expressed as an argument of the π operator. The τ operator transforms T to a Lambdapi type. t is introduced as a witness for the non emptiness of T. Thus, we get the following statement of which proof uses the exists introduction rule and the hypothesis.

```
symbol T : Set;

symbol thm0: π (`∀ (f : τ T → τ T), `∀ (t:τ T),(`∀ (x:τ T), `∀ (y:τ T), f x = f y)
                ⇒ (`∃ (c:τ T), `∀ (x:τ T), (f x = c))) ≔
begin
  assume f t h;
  apply (∃ᵢ (λ c, `∀ x, f x = c) (f t));
  assume x;
  apply (h x t)
end;
```

3 Event-B set theory in Lambdapi

B/Event-B [1] set theory is a typed-set theory. We will embed its constructs and its rules into Lambdapi.

3.1 Basic B type theory

First, we need to introduce in Lambdapi cartesian product, power set operators on types:

$\sigma ::=$	$\sigma\mathbb{P}\ \sigma$	power set
	$\mid \sigma\ \sigma\times\ \sigma$	cartesian product
	$\mid \sigma\text{BOOL} \mid \sigma\mathbb{Z}$	built-in boolean and integer types
	$\mid\ \sigma S$	for each user declared set S

This extensible inductive type is encoded in Lambdapi by the following declarations:

```
injective symbol σℙ: Set → Set; // power set of type
injective symbol σ×: Set → Set → Set; notation σ× infix left 24;// cartesian product of types
constant symbol σBOOL: Set;
constant symbol σℤ: Set;
constant symbol σS: Set; // user declared set S
```

3.2 Set operators

The classical set operators of Event-B derive from the declaration of the membership operator.

```
symbol ∈ [T:Set] : τ T → τ (σℙ T) → Prop;
```

Basic sets They are the maximal sets of elements of a given type. These maximal sets can be defined in a generic way. For this purpose, B introduces the `BIG` generic set.

```
constant symbol BIG [T:Set]: τ (σℙ T); // BIG
rule $x ∈ BIG ↪ ⊤; // BIG contains all elements of some type T
rule ℙ BIG ↪ BIG; // power set of BIG is also a maximal set
rule BIG × BIG ↪ BIG; // cartesian product of two maximal sets is maximal
```

We use BIG to define the sets of booleans and integers, as well as the user defined basic sets:

```
symbol BOOL : τ (σℙ σBOOL) ≔ BIG;
constant symbol TRUE : τ σBOOL;
constant symbol FALSE : τ σBOOL;

symbol ℤ : τ (σℙ σℤ) ≔ BIG;
... // operators, comparisons, min, max,...

symbol S: τ (σℙ σS) ≔ BIG;
```

Set constructors

```
rule $x ∈ ∅ ↪ ⊥;
...
constant symbol ∩ [T:Set] : τ (σℙ T) → τ (σℙ T) → τ (σℙ T); notation ∩ infix left 23;
rule $x ∈ $s1 ∩ $s2 ↪ $x ∈ $s1 ∧ $x ∈ $s2;
...
rule $e ∈ ℙ $S ↪ $e ⊆ $S;
...
symbol × [T1:Set] [T2:Set] :  τ (σℙ T1) → τ (σℙ T2) → τ (σℙ (T1 σ× T2));
constant symbol cset [T:Set]: (τ T → Prop) → τ (σℙ T); //set comprehension
```

3.3 Relational operators

Now we define relations from cartesian product, as $E_1 \leftrightarrow E_2$ is defined by $\mathbb{P}(E_1 \times E_2)$.

```
symbol rel (T1 T2: Set) ≔ τ (σℙ (T1 σ× T2));

injective symbol ↦ [T1:Set] [T2:Set] (x:τ T1) (y:τ T2) : τ (T1 σ× T2);

symbol ↔ [T1:Set] [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:Set] [T2:Set] : rel T1 T2 → τ (σℙ T1); notation dom prefix 30;
rule $x ∈ dom($r) ↪ `∃ y, $x ↦ y ∈ $r;

constant symbol ran [T1:Set] [T2:Set] : rel T1 T2 → τ (σℙ T2); notation ran prefix 30;
rule $y ∈ ran($r) ↪ `∃ x, x ↦ $y ∈ $r;
```

3.4 Functions

After that, we define functions, following Event-B formalism. As a reminder, we give the definitions of the functions we will need in the beginning of our example:

Meaning	Construct	Definition
partial functions	$f \in A \nrightarrow B$	{f . f ∈ A ↔ B ∧ ∀ x, ∀ y1, ∀ y2, x ↦ y1 ∈ f ∧ x ↦ y2 ∈ f ⇒ y1 = y2 }
partial surjective functions	$f \in A \twoheadrightarrow B$	{f . f ∈ (A ⇸ B) ∧ ran f = B }
total surjective functions	$f \in A \twoheadrightarrow B$	{f . (dom f) = A ∧ f ∈ A ⇸ B }

And here is the excerpt of the Lambdapi library needed to define surjective functions:

```
constant symbol ⇸ [T1:Set] [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);

constant symbol ⇻ [T1:Set] [T2:Set] :
    τ (σℙ T1) → τ (σℙ T2) → τ (σℙ (σℙ (T1 σ× T2))); notation ⇻ infix 11;
rule $f ∈ ($A ⇻ $B) ↪ ($f ∈ ($A ⇸ $B))  ∧  (ran $f = $B);

constant symbol → [T1:Set] [T2:Set] :
τ (σℙ T1) → τ (σℙ T2) → τ (σℙ (σℙ (T1 σ× T2))); notation → infix 11;
rule $f ∈ $A → $B ↪ (dom $f) = $A ∧ $f ∈ $A ⇻ $B;
```

4 Event-B deduction rules

Proofs in Event-B are driven by deduction rules. When giving a goal to prove to Rodin, it will apply some automated tactics. In case they fail, or if the user wants to guide the proof, he can choose the rules to apply, either transforming hypothesis or goal, or adding lemmas so the proof could be complete. The user can interact with Rodin by clicks on elements or by adding terms.

All these actions will appear in the proof tree of Rodin. However their granularity may vary from the application of basic proof rule to a call of an external tool. So far, we have only considered low level rules that have direct translations to Lambdapi proofs. This proof tree is in encoded in XML. We access it through a Java plug-in, to generate a Lambdapi file.

Each step of Rodin's proof tree has to be expressed in Lambdapi. With above extracts of library, we gave an idea of what is necessary to write expressions and predicates and we will now present how to express some deduction rules in Lambdapi.

Simple tactics First, we can simply use Lambdapi tactics, in the same way as Coq's tactics. The following arrays show the expression of some Rodin proof rules and the related Lambdapi's tactic:

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

Tactics defined as theorems Deduction rules will mainly be defined as theorems, using the command `symbol`, with the type-checking proof:

```
symbol Or2ImpGoal [P Q: Prop] : π (((¬ P) ⇒ Q) ⇒ (P ∨ Q)) ≔
begin
  assume P Q h;
  apply (λ h1 h2, ∨e P (¬ P) (P ∨ Q) h1 h2 (classic P))
  {assume hp;
   apply (∨i1 _ _ hp)
  }{assume hnp;
   apply (∨i2 _ _ (h hnp))
  }
end;
```

Remark: the brackets are used to signal optional arguments of an axiom.

This theorem can now be used, as in the following example, with the `apply` tactic of Lambdapi:

Rodin proof rule	Lambda-Pi tactic
$\dfrac{\Gamma \vdash \neg p \Rightarrow q}{\Gamma \vdash p \vee q}$	apply Or2ImpGoal

N-ary operators Some operators in Rodin are n-ary, while Lambdapi's operators are binary. The current choice is to transform n-ary expressions, with a plug-in Java included in Eclipse, in series of binary operators.

For instance, if the goal is $x \in E \cap F \cap G$, Rodin generates 3 sub-goals as the $\cap$ operator is seen as an n-ary operator. In Lambdapi, we macro generate a composition of introduction rules in order to get corresponding sub-goals:

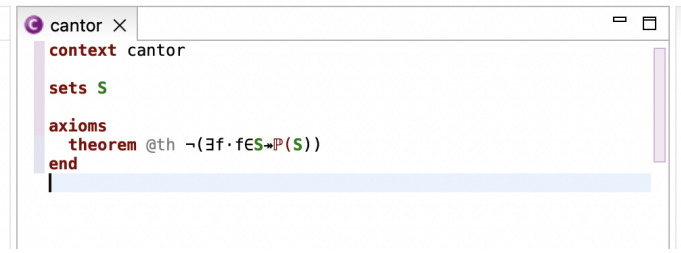
```
apply  (∧i (x ∈ E ∩ F) (x ∈ G)  ( ∧i (x ∈ E) (x ∈ F) _ _ ) _)
```

We note that $x \in A \cap B$ is equivalent to $x \in A \wedge x \in B$.

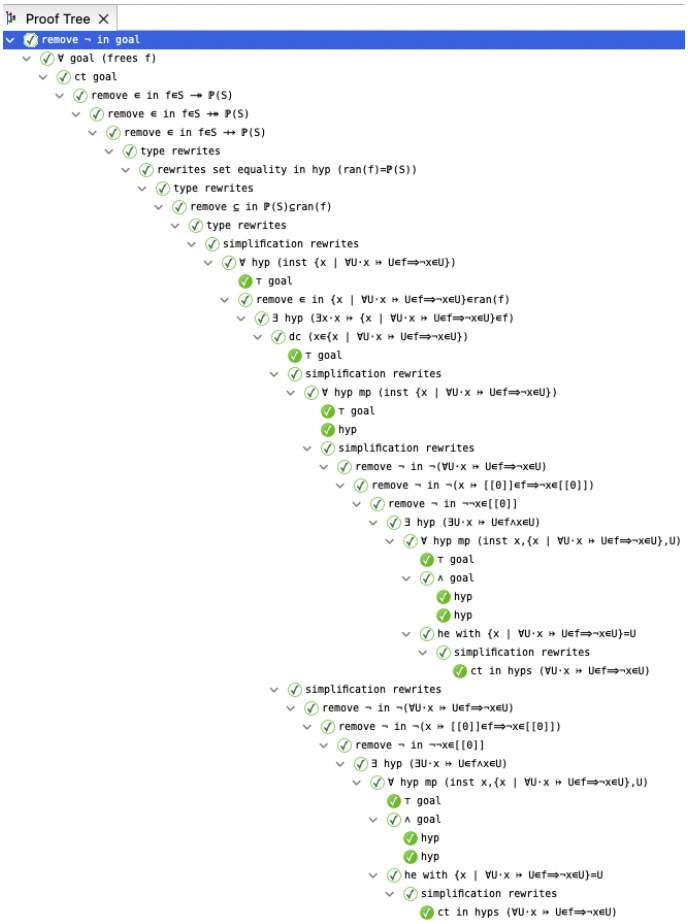
5 Cantor's theorem

To guide the beginning of our work, we focused on the proof of Cantor's theorem and we are able to check the translation of our proof with Lambdapi. A similar proof has been developed in TLA/TLAPS/VeriT/Lambdapi [3].

We made a strongly guided proof of Cantor's theorem, defined in the following context:



and obtained the following tree proof:



In the following lines we have the first steps of the translated proof. The gray boxes are not in the Lambdapi script, these are the rule details given by Rodin during the proof. In the comments (lines with //), we can read the nodes exported by Rodin and between the comments and the gray boxes, the lambdapi code.

```
require open lib.Set lib.Prop lib.FOL lib.Eq
    evb_ml.evb_ml evb_ml.evb_int evb_ml.evb_bool evbprf.evbprf;

constant symbol σS: Set;
symbol S: τ (σℙ σS) ≔ BIG;

symbol th'THM  : π (¬((`∃ (f: τ (σℙ (σS σ× (σℙ σS)))), f ∈ (S ⇸ (ℙ S))))) ≔
begin
  // rule: org.eventb.core.seqprover.rn
  // rewriting rule of Lambdapi
  simplify ¬; // useless in Lambdapi
  refine ((λ (__: π (`∀ (f: τ (σℙ (σS σ× (σℙ σS)))), ¬(f ∈ (S ⇸ (ℙ S))))), __) _);
  // syntactic identification following rewriting rules.
```

$$\vdash \neg(\exists f.f \in S \nrightarrow \mathbb{P}(S)) \xrightarrow{\text{remove } \neg \text{ in goal}} \vdash \forall f.\neg f \in S \nrightarrow \mathbb{P}(S))$$

```
  // rule: org.eventb.core.seqprover.allI
  assume  f; //Added free identifier f
```

$$\vdash \forall f.\neg f \in S \nrightarrow \mathbb{P}(S)) \xrightarrow{\forall \text{ goal (free f)}} \vdash \neg f \in S \nrightarrow \mathbb{P}(S)$$

```
// rule: org.eventb.core.seqprover.contrL1
apply not_intro [f ∈ (S ⇸ (ℙ S))];
```

$$\vdash \neg f \in S \nrightarrow \mathbb{P}(S) \xrightarrow{\text{ct goal}} f \in S \nrightarrow \mathbb{P}(S) \vdash \bot$$

```
assume  _H1; // f∈S ⇸ ℙ(S)
 // rule: org.eventb.core.seqprover.rmL1
apply (Generalize _H1); // f∈S ⇸ ℙ(S)
refine (and_imp _);
```

$$f \in S \nrightarrow \mathbb{P}(S) \vdash \bot \xrightarrow{\text{remove } \in \text{ in f}\in S\nrightarrow\mathbb{P}(S)} f \in S \twoheadrightarrow \mathbb{P}(S), dom(f) = S \vdash \bot$$

```
assume _H2; // dom(f)=S
assume _H3; // f∈S ⤔ ℙ(S)
// rule: org.eventb.core.seqprover.rmL1
apply (Generalize _H3); // f∈S ⤔ ℙ(S)
refine (and_imp _);
```

$$f \in S \twoheadrightarrow \mathbb{P}(S), dom(f) = S \vdash \bot \xrightarrow{\text{remove } \in \text{ in } f\in S\twoheadrightarrow\mathbb{P}(S)} f \in S \pfun \mathbb{P}(S), ran(f) = \mathbb{P}(S), dom(f) = S \vdash \bot$$

```
assume _H4; // f∈S ⇸ ℙ(S)
assume _H5; // ran(f)=ℙ(S)
// rule: org.eventb.core.seqprover.rmL1
apply (Generalize _H4); // f∈S ⇸ ℙ(S)
refine (and_imp _);
```

$$f \in S \pfun \mathbb{P}(S), ran(f) = \mathbb{P}(S), dom(f) = S \vdash \bot \xrightarrow{\text{remove } \in \text{ in } f\in S\pfun\mathbb{P}(S)}$$
$$f \in S \rel \mathbb{P}(S), \forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, ran(f) = \mathbb{P}(S), dom(f) = S \vdash \bot$$

```
assume _H6; // f∈S ⤔ ℙ(S)
assume _H7; // ∀x,x0,x1·x ↦ x0∈f∧x ↦ x1∈f⇒x0=x1
// rule: org.eventb.core.seqprover.typeRewrites
```

$$f \in S \rel \mathbb{P}(S), \forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, ran(f) = \mathbb{P}(S), dom(f) = S \vdash \bot$$
$$\xrightarrow{\text{type rewrites}} \forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, ran(f) = \mathbb{P}(S), dom(f) = S \vdash \bot$$

```
// rule: org.eventb.core.seqprover.setEqlRewrites
apply (set_eq_elim (ran f) (ℙ S) _ _H5); // ran(f)=ℙ(S)
refine (and_imp _);
```

$$ran(f) = \mathbb{P}(S) \xrightarrow{\text{rewrites set equality}} ran(f) \subseteq \mathbb{P}(S), \mathbb{P}(S) \subseteq ran(f)$$

```
assume _H8; // ran(f)⊆ℙ(S)
assume _H9; // ℙ(S)⊆ran(f)
 // rule: org.eventb.core.seqprover.typeRewrites
```

$$\forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, ran(f) \subseteq \mathbb{P}(S), \mathbb{P}(S) \subseteq ran(f), dom(f) = S \vdash \bot$$

$$\xrightarrow{\text{type rewrites}} \forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, \mathbb{P}(S) \subseteq ran(f), dom(f) = S \vdash \bot$$

```
// rule: org.eventb.core.seqprover.typeRewrites
have _H10: π ((`∀ (x: τ (σℙ σS)), (x ∈ (ℙ S) ⇒ x ∈ (ran f)))) { refine _H9 };
```

$$\mathbb{P}(S) \subseteq ran(f) \xrightarrow{\text{remove } \subseteq \text{ in } \mathbb{P}(S) \subseteq ran(f)} \forall x \cdot x \in \mathbb{P}(S) \Rightarrow x \in ran(f)$$

```
// rule: org.eventb.core.seqprover.typeRewrites
  have _H11 : π (`∀ (x: τ (σℙ σS)), (⊤ ⇒ x ∈ (ran f))) { // ∀x·⊤⇒x∈ran(f)
  refine _H10; // ∀x·x∈ℙ(S)⇒x∈ran(f)
};
```

$$\forall x \cdot x \in \mathbb{P}(S) \Rightarrow x \in ran(f) \xrightarrow{\text{type rewrites}} \forall x \cdot \top \Rightarrow x \in ran(f)$$

```
// rule: org.eventb.core.seqprover.autoRewritesL4
 have _H12 : π ((`∀ (x: τ (σℙ σS)), x ∈ (ran f))) { // ∀x·x∈ran(f)
   apply (Generalize (rsimpl_intro _H11)); // ∀x·⊤⇒x∈ran(f)
   refine (λ p,p); // ∀x·⊤⇒x∈ran(f)
 };
```

$$\forall x \cdot \top \Rightarrow x \in ran(f) \xrightarrow{\text{simplification rewrites}} \forall x \cdot x \in ran(f)$$

```
// rule: org.eventb.core.seqprover.allD
apply (Generalize ((_H12 (`cset (x: τ σS), (`∀ (U: τ (σℙ σS)), ((x ↦ U) ∈ f ⇒ x ∉ U))))));
```

$$\forall x \cdot x \in ran(f) \xrightarrow{\forall \text{ hyp instanciated with}} \{x | \forall U \cdot x \mapsto U \in f \Rightarrow x \notin U\} \in ran(f)$$

Antecedent 1 (OP): $\vdash \top$

Antecedent 2:

$$\forall x, x0, x1 \cdot x \mapsto x0 \in f \land x \mapsto x1 \in f \Rightarrow x0 = x1, \{x | \forall U \cdot x \mapsto U \in f \Rightarrow x \notin U\} \in ran(f), dom(f) = S \vdash \bot$$

...

6 Conclusion

There is still a significant work ahead to perform the complete translation of Event-B proofs to Lambdapi.

The encoding is going to evolve to take into account all the peculiarities of Event-B, Rodin and Lambdapi. We have also to complete the translation of all the deduction's rules of Rodin.

Another point is that the automated proofs are a strength of Rodin. They are provided by integrated solvers or external SMT solvers. We have to implement the treatment made by Rodin for SMT solvers. It begins by removing all set-theoretical constructs in the formulas before sharing them with the SMT solvers. The SMT solver gives its result and sometimes useful data to Rodin. In the future, we will look for

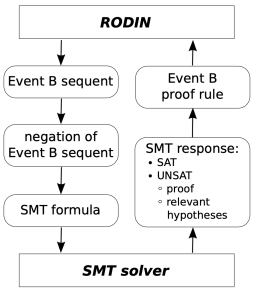


Figure 1: Rodin and SMT solvers[6].

an appropriate way to translate this information of the solver in Lambdapi.

Last, we will have to express the full Event-B, define machines, events, refinements and the mechanisms of proof obligations, some of the major features of the formal method Event-B.

References

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

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

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

[4] Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016.

[5] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo, 2023.

[6] David Déharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. Integrating smt solvers in rodin. *Science of Computer Programming*, 94:130–143, 2014. Abstract State Machines, Alloy, B, VDM, and Z.