Verification of Event-B proofs through their translation to Lambdapi

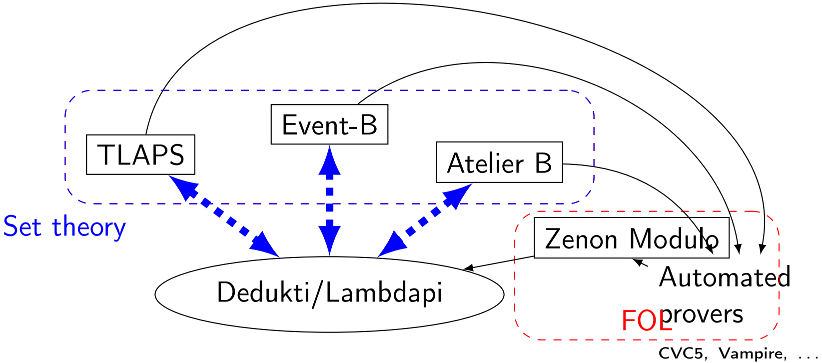
Jean-Paul Bodeveix, Anne Grieu

INP - IRIT
Université de Toulouse
Équipe ACADIE

June 2025

Context

Context - ICSPA [1] Project : Dedukti/Lambdapi as pivot language



ICSPA Partners

- SAMOVAR
- INRIA Nancy
- INRIA Paris-Saclay
- IRIT
- LIRMM
- CLEARSY

1. https://icspa.inria.fr/fr/

Event-B to Lambdapi

Goal

- Transform an Event-B **proof tree** to a Lambdapi script.
- Verification of the proof by Lambdapi.

Issues

- Describe the mathematical language of Event-B
- Describe rewrite rules and deduction rules of Event-B
- Take into account features of Rodin proof framework
- Build a faithful (parallel) trace of the Rodin proof tree in the Lambdapi script.

Event-B to Lambdapi

Lambdapi : Logical framework based on $\lambda\Pi$-Calculus Modulo Theory

« Lambdapi is an interactive proof system featuring ***dependent types*** *like in Martin-Lőf's type theory, but allowing to define objects and types using* ***oriented equations****, aka* ***rewriting rules****, and reason modulo those equations. »* [2]

$\lambda\Pi$ terms

$t\ , t'$ ::=	V	variable
	$\mid \texttt{TYPE}$	sort for types
	$\mid \Pi\ (V : t),\ t'$	dependent product type
	$\mid \lambda\ (V : t),\ t'$	abstraction
	$\mid t\ \ t'$	application
	$\mid t \rightarrow t'$	abbreviation for $\Pi\ (V : t),\ t'$ when $V \notin t'$

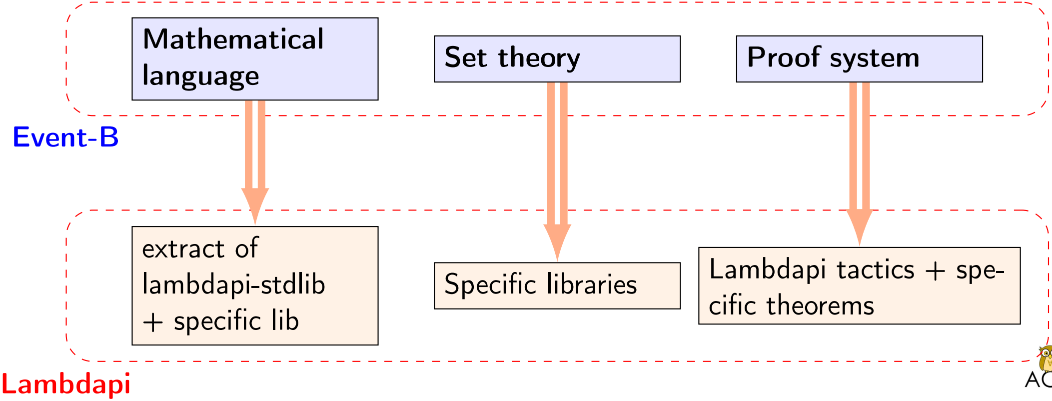
Rules

r	::=	$t \hookrightarrow t'$	reasoning modulo rewriting rules

2. https://lambdapi.readthedocs.io/en/latest/about.html

Embedding Event-B in Lambdapi

Event-B theory expressed in Lambdapi, so we can check proofs based on this theory with Lambdapi.



First order logic [3]

The mathematical language of Event-B is based on first order classical logic.
We manage to use part of the lambdapi-stdlib, a library written to cover a large part of common logics, to express a logic similar to the one of Event-B.

Propositional logic

```
constant symbol Prop : TYPE;
// Associates a type of a proof to a proposition
injective symbol π : Prop → TYPE;
```

Types of datatypes

```
constant symbol Set : TYPE;
// Associates a type to a datatype
injective symbol τ : Set → TYPE;
```

3. Standard library : `https://github.com/Deducteam/lambdapi-stdlib`

Axiomatisation of first order logic - some excerpt

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;
```

Implication (Coq style)

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

Classical logic - axiom

```
symbol em p : π (p ∨ ¬ p); // excluded middle
```

Related sequents for conjunction

$$\frac{\Gamma \vdash p \quad \Gamma \vdash q}{\Gamma \vdash p \wedge q}\ (\wedge_i)$$

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p}\ (\wedge_{e1})$$

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q}\ (\wedge_{e2})$$

ACADIE

Event-B set theory

Event-B types :

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

In lambdapi :

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

Set operators

Classical set operators of Event-B derive from membership operator :

```
symbol ∈ [T:Set] : τ T → τ (σℙ T) → Prop;
// extensionnality axiom
symbol ext [T] (e1 e2: τℙ T): π (`∀ x, x ∈ e1 ⇔ x ∈ e2) → π (e1 = e2);
```

Generic maximal set BIG

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

ACADIE

Derived operators

```
// derived set operators with rules
constant symbol ∩ [T1 T2:Set]: (τ T1 → τℙ T2) → τℙ T2;
rule $x ∈ $s1 ∩ $s2 ↪ $x ∈ $s1 ∧ $x ∈ $s2;
//pairs
injective symbol ↦ [T1 T2:Set] (x: τ T1) (y: τ T2): τ (T1 σ× T2);
// binary relations
symbol ↔ [T1 T2:Set] (A: τ (σℙ T1)) (B: τ (σℙ T2)): τ (σℙ (σℙ (T1 σ× T2)
    )) ≔ ℙ (A × B);    notation ↔ infix 11;
// domain
constant symbol dom [T1 T2:Set]: τ (σℙ (T1 σ× T2)) → τ (σℙ T1);
notation dom prefix 30;
rule $x ∈ dom($r) ↪ `∃ y, $x ↦ y ∈ $r;
// ran, relations, partial functions, total functions
constant symbol ⇸ [T1 T2:Set]: τℙ T1 → τℙ T2 → τℙ (σℙ (T1 σ× T2));
notation ⇸ infix 11;
rule $f ∈ $A → $B ↪ (dom $f) = $A ∧ $f ∈ $A ⇸ $B;
```

From an Event-B proof tree to a Lambdapi proof script

ACADIE

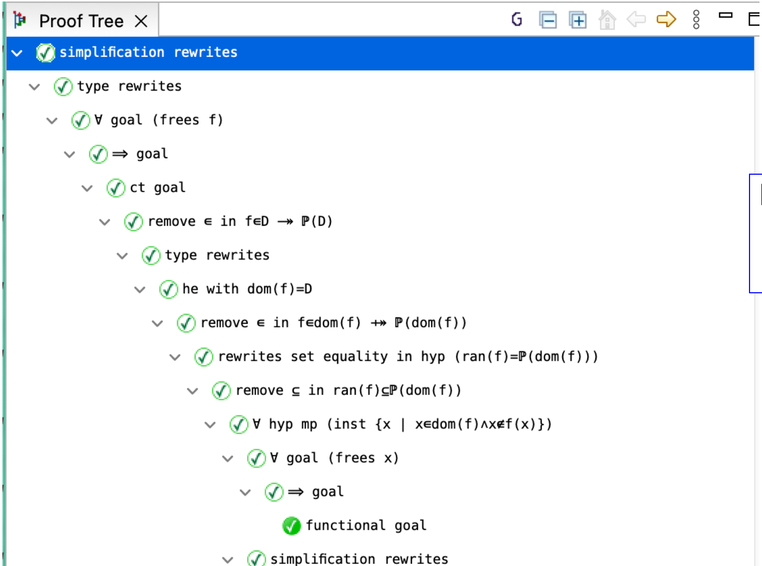
Event-B proof system

Event-B

- Inference rules and rewriting rules on hypotheses or goal
- Simplification rules
- Introduction of lemmas.
- Combination of rules (GenMP, ...)
- Call to internal and external provers

ACADIE

Translation of Rodin proof tree

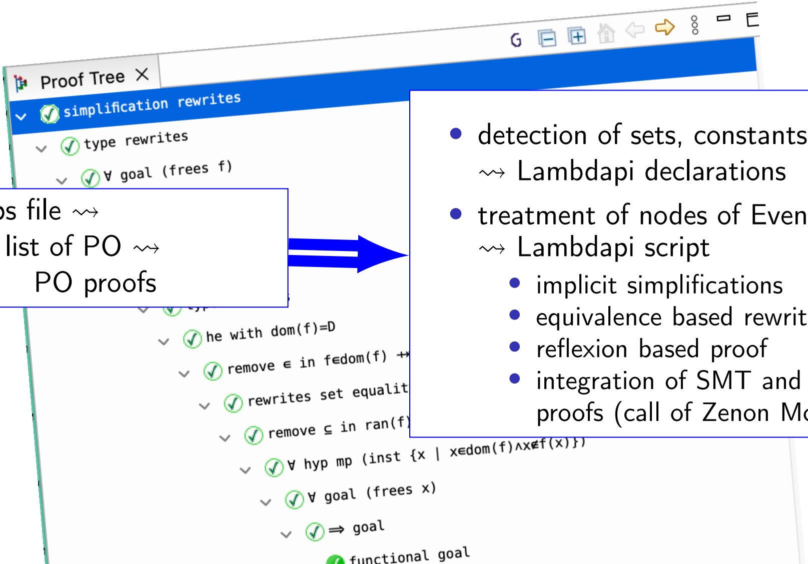


bps file ⇝

list of PO ⇝

PO proofs

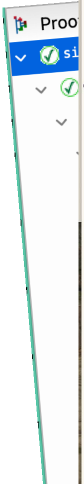
Translation of Rodin proof tree



bps file ⇝
list of PO ⇝
PO proofs

- detection of sets, constants in the proof tree
  ⇝ Lambdapi declarations
- treatment of nodes of Event-B proof tree
  ⇝ Lambdapi script
  - implicit simplifications
  - equivalence based rewriting
  - reflexion based proof
  - integration of SMT and internal provers proofs (call of Zenon Modulo)

Translation of Rodin proof tree




```
require open lib.Set lib.Prop lib.FOL lib.Eq evb_ml.evb_ml evb_ml.evb_int evb_ml.evb_bool evbprf.evbprf meta.inhabited zenon.zenon;
require meta.btyprod meta.allprod meta.exprod;
// types
constant symbol σS: Set;
constant symbol σS_elem : τ σS; // not empty
symbol S: τℙ σS ≔ BIG;
symbol ax_in_S: π (`∀ (X: τ σS), X ∈ S) ≔ λ _, ⊤ᵢ;

symbol fct'WD (D: τ (σℙ σS)): π ((`∀ (f0q: τ (σℙ (σS σ× (σℙ σS)))), `∀ (x1q: τ σS), `∀ (y2q: τ (σℙ σS))), ((f0q ∈ (D ⇸ (ℙ D)) ∧ x1q ∈ D ∧ (x1q ↦ y2q) ∈ f0q) ⇒ (x1q ∈ (dom f0q) ∧ f0q ∈ (S ⇸ (ℙ S)))))) ≔
begin
  assume D; // constants
  // rule: org.eventb.core.seqprover.typeRewrites
  //type rewrites
  // # ants: 1
  // goal: ∀f,x,y·f∈D ⇸ ℙ(D)∧x∈D∧x ↦ y∈f⇒x∈dom(f)∧f∈S ⇸ ℙ(S)
  // new goal: null
  // rule: org.eventb.core.seqprover.allI
  assume _f x y;
  // rule: org.eventb.core.seqprover.impI
  refine (and_imp _);
  assume _H1; // f∈D ⇸ ℙ(D)
  refine (and_imp _);
  assume _H2; // x∈D
  assume _H3; // x ↦ y∈f
  // rule: org.eventb.core.seqprover.conj
  apply (∧ᵢ _ _)
  {
    // rule: org.eventb.core.seqprover.rmL2
      refine ((λ _: π ((`∃ (x00q: τ (σℙ σS)), (x ↦ x00q) ∈ f)), _) _); //∃x0·x ↦ x0∈f
    // rule: org.eventb.core.seqprover.exI
    have _H4 : π (⊤) { // ⊤
      // rule: org.eventb.core.seqprover.trueGoal
      refine ⊤ᵢ;
    }
```

Details of the proof - All intro

```
!symbol fct'WD (D: τ (σℙ σS)): π ((`∀ (f0q: τ (σℙ (σS σ× (σℙ σS)))), `∀ (x1q: τ σS), `∀ (y2q: τ (σℙ σS
)), ((f0q ∈ (D ⇸ (ℙ D)) ∧ x1q ∈ D ∧ (x1q ↦ y2q) ∈ f0q) ⇒ (x1q ∈ (dom f0q) ∧ f0q ∈ (S ⇸ (ℙ S)))))) ≔
begin
!  assume D; // constants
   // rule: org.eventb.core.seqprover.typeRewrites
   //type rewrites
   // # ants: 1
   // goal: ∀f,x,y·f∈D ⇸ ℙ(D)∧x∈D∧x ↦ y∈f⇒x∈dom(f)∧f∈S ⇸ ℙ(S)
   // new goal: null
   // rule: org.eventb.core.seqprover.allI
!  assume  f x y;
   // rule: org.eventb.core.seqprover.impI
!  refine (and_imp _);
- 20k  gfCantor.lp  LambdaPi                                          LambdaPi  unix | 11:10   1
: (←)(→) *lp-logs* ×  *GNU Emacs* ×  *EGLOT (evb2lp_std/(lambdapi-mode)) events* ×  *Quail Completions* ×  *Goals*
D: τ (σℙ σS)

?27: π (`∀ f0q, `∀ x1q, `∀ y2q, ((f0q ∈ (D ⇸ ℙ D)) ∧ ((x1q ∈ D) ∧ ((x1q ↦ y2q) ∈ f0q))) ⇒ ((x1q ∈ dom f0q) ∧ (f0q ∈
(S ⇸ ℙ S))))
```

$$\frac{\mathbf{H} \vdash \mathbf{P(x)}}{\mathbf{H} \vdash \forall \mathbf{x} \cdot \mathbf{P(x)}}$$

Details of the proof - All intro

```
symbol fct'WD (D: τ (σℙ σS)): π ((`∀ (f0q: τ (σℙ (σS σ× (σℙ σS)))), `∀ (x1q: τ σS), `∀ (y2q: τ (σℙ σS
)), ((f0q ∈ (D ⇸ (ℙ D)) ∧ x1q ∈ D ∧ (x1q ↦ y2q) ∈ f0q) ⇒ (x1q ∈ (dom f0q) ∧ f0q ∈ (S ⇸ (ℙ S)))))) ≔
begin
  assume D; // constants
  // rule: org.eventb.core.seqprover.typeRewrites
  //type rewrites
  // # ants: 1
  // goal: ∀f,x,y·f∈D ⇸ ℙ(D)∧x∈D∧x ↦ y∈f⇒x∈dom(f)∧f∈S ⇸ ℙ(S)
  // new goal: null
  // rule: org.eventb.core.seqprover.allI
  assume  f x y;
  // rule: org.eventb.core.seqprover.impI
  refine (and_imp _);
  assume _H1; // f∈D ⇸ ℙ(D)
- 20k  gfCantor.lp  LambdaPi                                    LambdaPi  unix | 18:15   1

: (←)(→) *lp-logs* ×  *GNU Emacs* ×  *EGLOT (evb2lp_std/(lambdapi-mode)) events* ×  *Quail Completions* ×  *Goals*

D: τ (σℙ σS)
f: τ (σℙ (σS σ× σℙ σS))
x: τ σS
y: τ (σℙ σS)

?32: π (((f ∈ (D ⇸ ℙ D)) ∧ ((x ∈ D) ∧ ((x ↦ y) ∈ f))) ⇒ ((x ∈ dom f) ∧ (f ∈ (S ⇸ ℙ S))))
```

$$\frac{\mathbf{H} \vdash \mathbf{P(x)}}{\mathbf{H} \vdash \forall \mathbf{x} \cdot \mathbf{P(x)}}$$

Details of the proof - Conjunction intro

```
refine (and_imp _ _);
assume _H2; // x∈D
assume _H3; // x ↦ y∈f
// rule: org.eventb.core.seqprover.conj
apply (∧ᵢ _ _ _ _)
{
  // rule: org.eventb.core.seqprover.rmL2
  refine ((λ _ : π ((`∃ (x00g: τ (σℙ σS)), (x ↦ x00g) ∈ f)), _ ) _); //∃x0·x ↦ x0∈f
  // rule: org.eventb.core.seqprover.exI
  have _H4 : π (⊤) { // ⊤
    // rule: org.eventb.core.seqprover.trueGoal
    refine ⊤ᵢ;
```

$$\frac{H \vdash P \qquad H \vdash Q}{H \vdash P \land Q}$$

```
- 20k  gfCantor.lp  LambdaPi                                    LambdaPi  unix | 24:12
: (←)(→) *lp-logs* ×  *GNU Emacs* ×  *EGLOT (evb2lp_std/(lambdapi-mode)) events* ×  *Quail Completions* ×  *Goals
D: τ (σℙ σS)
f: τ (σℙ (σS σ× σℙ σS))
x: τ σS
y: τ (σℙ σS)
_H1: π (f ∈ (D ⇸ ℙ D))
_H2: π (x ∈ D)
_H3: π ((x ↦ y) ∈ f)

?49: π ((x ∈ dom f) ∧ (f ∈ (S ⇸ ℙ S)))
```

Details of the proof - Conjunction intro

$$\frac{H \vdash P \qquad H \vdash Q}{H \vdash P \wedge Q}$$

```
// rule: org.eventb.core.seqprover.conj
apply (∧ᵢ _ _)
{
  // rule: org.eventb.core.seqprover.rmL2
    refine ((λ _: π ((`∃ (x00q: τ (σℙ σS)), (x ↦ x00q) ∈ f)), _) _); //∃x0·x ↦ x0∈f
  // rule: org.eventb.core.seqprover.exI
  have _H4 : π (⊤) { // ⊤
    // rule: org.eventb.core.seqprover.trueGoal
    refine ⊤ᵢ;
  };
  apply (∃ᵢ y);
  // rule: org.eventb.core.seqprover.hyp
```

```
- 20k  gfCantor.lp  LambdaPi                LambdaPi  unix | 27: 2  4
```

```
: (←)(→) *lp-logs* ×  *GNU Emacs* ×  *EGLOT (evb2lp_std/(lambdapi-mode)) events* ×  *Quail Completions* ×  *Goals
```

```
D: τ (σℙ σS)
f: τ (σℙ (σS σ× σℙ σS))
x: τ σS
y: τ (σℙ σS)
_H1: π (f ∈ (D ↠ ℙ D))
_H2: π (x ∈ D)
_H3: π ((x ↦ y) ∈ f)

?56: π (x ∈ dom f)
```

Details of the proof - Conjunction intro

```
!  apply (∧ᵢ _ _)
   {
     // rule: org.eventb.core.seqprover.rmL2
!      refine ((λ _: π ((`∃ (x00q: τ (σℙ σS)), (x ↦ x00q) ∈ f)), _) _); //∃x0·x ↦ x0∈f
     // rule: org.eventb.core.seqprover.exI
!    have _H4 : π (⊤) { // ⊤
       // rule: org.eventb.core.seqprover.trueGoal
!      refine ⊤ᵢ;
     };
!    apply (∃ᵢ y);
     // rule: org.eventb.core.seqprover.hyp
!    refine _H3; // x ↦ y∈f
   }
   {
     // rule: org.eventb.core.seqprover.isFunGoal
!    refine (partialFunFun D (ℙ D) f _H1); // f∈D ⇸ ℙ(D)
   }
!end;
```

$$\frac{H \vdash P \qquad H \vdash Q}{H \vdash P \wedge Q}$$

```
- 20k  gfCantor.lp  LambdaPi                               LambdaPi  unix | 39: 2  5
: (←)(→) *lp-logs* ×  *GNU Emacs* ×  *EGLOT (evb2lp_std/(lambdapi-mode)) events* ×  *Quail Completions* ×  *Goals
D: τ (σℙ σS)
f: τ (σℙ (σS σ× σℙ σS))
x: τ σS
y: τ (σℙ σS)
_H1: π (f ∈ (D ⇸ ℙ D))
_H2: π (x ∈ D)
_H3: π ((x ↦ y) ∈ f)

?57: π (f ∈ (S ⇸ ℙ S))
```

ACADIE

Types, constants, hypotheses

First step through the proof tree

The plug-in extracts declared sets, constants, hypotheses that are available in the proof tree to declare them in the Lambdapi script.

Hypotheses

The plug-in creates a mapping between Event-B hypotheses and Lambdapi identifiers.

Types

```
public void generate(IPRProof p) throws RodinDBException, CoreException {
    out.println("//␣types");
    sets.clear();
    List<String> csts = new LinkedList<>();
    for (String s : p.getSets()) {
      sets.add(s);
      if (knownSets.contains(s)) continue;
      knownSets.add(s);
      out.println("constant␣symbol␣σ"+s+":␣Set;");
      out.println("constant␣symbol␣σ"+s+"_elem␣:␣τ␣σ"+s+";␣//␣not␣empty");
      out.println("symbol␣"+s+":␣τℙ␣σ"+s+"␣≔␣BIG;");
      out.println("symbol␣ax_in_"+s+":␣π␣(`∀␣(X:␣τ␣σ"+s+"),␣X␣∈␣"+s+")␣≔␣λ␣_,␣⊤ᵢ;");
    }
    ...
```

ACADIE

Proof nodes

Proof tree

Recursive path through the proof tree.
Syntactic research with the name of the rule of the node and the plug-in apply the right processing.

Example of processing

- True goal in Event-B corresponds to applying $\top_i : \pi\top$ constructor in Lambdapi.
- To split a n-ary conjunction : apply a proof-term schema :

```
refine (∧ᵢ [P₁ ∧ (P2 ∧ P3)] _ ( ∧ᵢ [P2] [P3] _ _) ){...}{...}{...};
```

ACADIE

Simple rules

True goal

```
} else if (rn.startsWith("⊤␣goal")) {
  out.println(tab+"refine␣⊤ᵢ;");
}
```

Split conjunction

```
} else if (rn.startsWith("∧␣goal")) {
  out.println(tab+"apply␣"+genSplit(g,g.getTag()));
  for (IProofTreeNode c : children) {
    out.println(tab+"{");
    generate(hnum, tab+"␣␣",p,pt,c);
    out.println(tab+"}");
  }
}
```

Remove membership

Set operators defined with rule in Lambdapi upon ∈. Deal with :

- Event-B automatically split the conjunctions in the terms
- keep a trace of the Event-B step in the Lambdapi script when a `rule` is used

```
} else if (rn.startsWith("remove␣∈␣in␣goal")) {
  if (children.length > 1) {
    Expression rhs = ((RelationalPredicate)g).getRight();
    out.println(tab+"apply␣"+genSplit(rhs,rhs.getTag()));
    for (IProofTreeNode c : children) {
      out.println(tab+"{");
      generate(hnum, tab+"␣␣",p,pt,c);
      out.println(tab+"}");
    }
  } else {
    Predicate ng = children[0].getSequent().goal();
    out.println(tab+"␣␣refine␣((λ␣_ _:␣π␣("+Formula2LP.translate(ng)+"),␣_ _)␣_);␣//" + ng);
    generate(hnum, tab,p,pt, children[0]);
  }
```

ACADIE

Proof by reflexion

Eliminate quantification over product types

Meta theorems are proved and instanciated with the help of the plug-in.

```
} else if (rn.startsWith("remove ⊆ in goal")) {
    Expression e1 = (Expression) g.getChild(0);
    Expression e2 = (Expression) g.getChild(1);
    Predicate ng = children[0].getSequent().goal();
    Type t = e1.getType().getBaseType();
    if (t instanceof ProductType) {
        String lam = "(λ __, meta.allprod.NotAll (__ ∈ ("+Formula2LP.translate(e1)+ ") ⇒ 
    __ ∈ ("+Formula2LP.translate(e2)+")))";
        out.println(tab+"refine (( let ___ : π (("+Formula2LP.translate(ng)+") ⇒ ("+
    Formula2LP.translate(g)+ ")) ≔ ∧e2 (meta.allprod.elimAllProd_eqv "+toBProd(t)+" "+lam
    +") in ___) _);");
  }
  generate(hnum, tab,p,pt, children[0]);
}
```

A more complex example

Simplification rewrites [4]

Lots of simplification rules in Event-B.

Three ways

- Use Lambdapi rewrite rules $\rightsquigarrow$ no proof information
- Equivalence rewriting (setoid rewrite) with Plug-in generated proofs
- Equivalence rewriting (setoid rewrite) with Lambdapi generated proofs

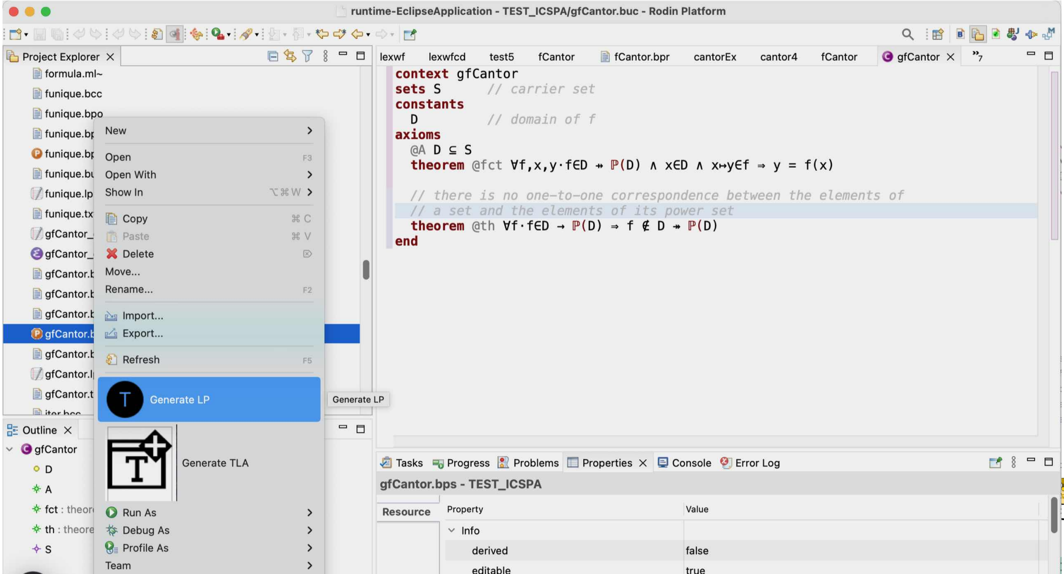
4. https://wiki.event-b.org/index.php/All_Rewrite_Rules

Simplification rewrites 1/3

```
else if (rn.startsWith("simplification␣rewrites")) {
  assert (children.length == 1);
  IAntecedent h = r.getAntecedents()[0];
  int i = 0;
  for (IHypAction a : h.getHypActions()) {
        if (!(a instanceof IRewriteHypAction)) continue;
        IRewriteHypAction rw = (IRewriteHypAction) a;
        for (Predicate hyp: rw.getHyps()) {
            Function<Predicate,Result<Predicate>> S = genRepeat(genSimplify);
            Result<Predicate> R = S.apply(hyp);
            String tac = R.tac(); // get equivalence proof
            Predicate nhyp = R.pred(); //get simplified predicate
            Predicate rdhyp=hyp.rewrite(REWRITER);
            if (!rdhyp.equals(nhyp)) ... // divergence of the proof tree
```

ACADIE

GenerateLP


runtime-EclipseApplication - TEST_ICSPA/gfCantor.buc - Rodin Platform
Project Explorer
formula.ml~
funique.bcc
funique.bpo
gfCantor
lexwf
lexwfcd
test5
fCantor
fCantor.bpr
cantorEx
cantor4
fCantor
gfCantor
context gfCantor
sets S // carrier set
constants
D // domain of f
axioms
@A D ⊆ S
theorem @fct ∀f,x,y·f∈D ⇸ ℙ(D) ∧ x∈D ∧ x↦y∈f ⇒ y = f(x)
// there is no one-to-one correspondence between the elements of
// a set and the elements of its power set
theorem @th ∀f·f∈D → ℙ(D) ⇒ f ∉ D ↠ ℙ(D)
end
New
Open F3
Open With
Show In
Copy
Paste
Delete
Move...
Rename... F2
Import...
Export...
Refresh F5
Generate LP
Generate LP
Generate TLA
Run As
Debug As
Profile As
Team
Outline
gfCantor
D
A
fct : theor
th : theore
S
Tasks Progress Problems Properties Console Error Log
gfCantor.bps - TEST_ICSPA
Resource
Property
Value
Info
derived
false
editable
true

Conclusions

- Following Rodin proofs is hard – undocumented side effects (e.g. flattening of associative operators), large set of simplification rules and tactics
- Generation of proof terms through the plug-in
- Generation of proof terms through Lambdapi
- Proofs by reflexion

Missing

- language to express rules and automatic translators to LambdaPi (should be part of Rodin...)
- automatic translation of Rodin internal rules
- dedicated lambdaPi tactics

ACADIE