Karnaj

Zeste de savoir

Des arbres de preuve en LaTeX

10 février 2021

Table des matières

Lorsqu'on fait de la logique par exemple, il arrive de vouloir écrire des arbres de preuve. Plusieurs *packages* LaTeX permettent de mettre en forme de tels arbres. Ici nous retiendrons bussproofs et ebproof qui nous permettent d'écrire les arbres de cette manière.

```latex
% Avec ebproof
\begin{prooftree}
      \infer0[$\text{ax}$]{A, \neg A \vdash A}
      \infer0[$\text{ax}$]{A, \neg A \vdash A \implies \bot }
   \infer2[$\implies_\text{e}$]{A, \neg A \vdash \bot }
\infer1[$\bot_\text{e}$]{A, \neg A \vdash B}
\end{prooftree}

% Avec bussproofs
\begin{prooftree}
      \AxiomC{}
      \RightLabel{$\text{ax}$}
      \UnaryInfC{$A, \neg A \vdash A$}
      \AxiomC{}
      \RightLabel{$\text{ax}$}
      \UnaryInfC{$A, \neg A \vdash A \implies \bot$}
   \RightLabel{$\implies_\text{e}$}
   \BinaryInfC{$A, \neg A \vdash \bot$}
\RightLabel{$\bot_\text{e}$}
\UnaryInfC{$A, \neg A \vdash B$}
\end{prooftree}
```

Qui nous donne quelque chose qui ressemble à

$$\dfrac{\dfrac{}{A, \neg A \vdash A}\text{ax} \qquad \dfrac{}{A, \neg A \vdash A \implies \bot}\text{ax}}{\dfrac{A, \neg A \vdash \bot}{A, \neg A \vdash B}\bot_\text{e}} \implies_\text{e}$$

obtenu avec le code LaTeX suivant.

```
\dfrac{
   \dfrac{}{A, \neg A \vdash A} \text{ax} \qquad
   \dfrac{}{A, \neg A \vdash A \implies \bot} \text{ax}
}{
   \dfrac{A, \neg A \vdash \bot}{A, \neg A \vdash B}
       \bot_{\text{e}}
} \implies_{\text{e}}
```

Autant dire qu'on est content d'avoir de *packages* qui nous aident à la mise en forme de tels arbres! Néanmoins, la syntaxe de ces *packages* n'est pas non plus la plus simple (et celle de `bussproofs` est encore plus verbeuse).

Notre mission est simple. Nous allons simplifier l'écriture de telles preuves à l'aide de Ruby.

Auteur fou?

Ce billet a été écrit totalement en vrac, parce que ça fait quelques jours que je procrastine son écriture parce que je ne voulais pas trop organiser ce que j'allais écrire! J'ai donc attendu d'avoir autre chose à faire, histoire de procrastiner cette autre chose en écrivant ce billet (sans avoir d'abord organisé ce que j'allais écrire).

1. Représenter les arbres

Avant toute chose, il faudra voir comment représenter les arbres de preuve. Nous créons une classe pour cela. On construit un arbre de preuve en donnant des prémisses et une conclusion. Nous rajoutons également la règle utilisée pour passer des prémisses à la conclusion. Ici nous l'appelons `label`, car nous voulons plus savoir ce qui doit être affiché que la règle utilisée.

```ruby
class ProofTree
  def initialize(premises, conclusion, label)
    @premises = premises.dup
    @conclusion = conclusion
    @label = label
  end

  def [](index)
    @premises[index]
  end

  def format(formatter)
    formatter.format(self)
  end
end
```

Nous avons une méthode `[]` pour accéder aux prémisses d'un arbre et une méthode `format` qui prend un formateur en entrée et l'utilise pour formater l'arbre. Par exemple, on pourrait avoir un objet qui donne un format texte, un autre qui donne un format ebproof, un autre pour le format bussproofs, etc.

Au niveau de la représentation des labdels, on pourrait faire beaucoup de choses. Nous allons rester simple, une label sera un objet avec deux chaînes de caractères: celle à utiliser dans le cas d'un formatage en texte, et l'autre dans le cas d'un formatage en LaTeX.

```ruby
class Label
  attr_reader :text, :latex

  def initialize(text, latex)
    @text = text
    @latex = latex
  end
end

Nope = Label.new("", "")
Axiom = Label.new("axiom", "\\text{ax}")
IntroTrue = Label.new("intro_true", "\\top_\\text{i}")
ElimFalse = Label.new("elim_false", "\\bot_\\text{e}")
ElimImpl = Label.new("elim_impl", "\\implies_\\text{e}")
IntroImpl = Label.new("intro_impl", "\\implies_\\text{i}")
ElimAnd = Label.new("elim_and", "\\land_\\text{e}")
IntroAnd = Label.new("intro_and", "\\land_\\text{i}")
ElimOr = Label.new("elim_or", "\\lor_\\text{e}")
IntroOr = Label.new("intro_or", "\\lor_\\text{i}")
```

Nous avons notre classe pour les labels, et nous en avons instancié plusieurs qui correspondent à des règles classiques de logique. Le label `Nope` sera utilisée lorsque nous ne voulons pas de label. D'ailleurs, nous allons le mettre comme valeur par défaut du paramètre `label` du constructeur de `ProofTree`.

Nous pouvons alors créer un arbre de la sorte.

```ruby
tree = ProofTree.new([], "A, \neg A \vdash A", Axiom)
```

Pour le moment, la syntaxe est toujours compliquée, dans la suite, nous allons faire en sorte d'avoir une meilleure syntaxe, et nous allons bien sûr écrire du code pour le formatage!

2. Un petit DSL

Pour taper des preuves plus facilement, nous allons écrire quelque chose qui ressemblera à un petit DSL. Pour comprendre l'idée du DSL et du code qui va être écrit, cet article de Synbioz est plutôt sympa. Nous allons essayer d'avoir une syntaxe de ce genre.

```
proof = Proof.new do
  infer 'A, \neg A \vdash B', with: ElimFalse do
    infer 'A, \neg A \vdash \bot', with: ElimImpl do
      infer 'A, \neg A \vdash A \implies \bot', with: Axiom
      infer 'A, \neg A \vdash A', with: Axiom
    end
  end
end
```

Ceci correspond à l'arbre de preuve présenté en introduction. Ici, la méthode `infer` prend en paramètre l'élément que l'on veut prouver, le label et un bloc correspondant à ce qu'on fait pour le prouver (aux sous-arbres de preuves utilisés). Ainsi, on peut lire `infer text with: rule block` comme «on infère `text` en utilisant la règle `rule` sur les sous-arbres de `block`.

Écrivons la classe `Proof` correspondant à cela. Ella un constructeur qui prend en paramètre un bloc, et elle a également une méthode `infer`.

```
class Proof
  attr_reader :tree

  def initialize(&block)
    raise ArgumentError, "need block to create proof" unless
        block_given?

    @children = Hash.new { |h, k| h[k] = [] }
    @depth = 0
    @tree = instance_eval(&block)
  end

  private

  def infer(text, with: Nope)
    if block_given?
      @depth += 1
      yield
      @depth -= 1
    end
    tree = ProofTree.new(@children[@depth + 1], text, with)
    @children[@depth + 1] = []
    @children[@depth] << tree
    tree
  end
end
```

Une preuve a un arbre de preuve `@tree` qu'on construit grâce au bloc donné au constructeur de `Proof`. Dans la méthode `infer`, il faut construire les arbres de preuve en faisant attention aux liens de parenté.

```ruby
infer 'A' do
  infer 'A.1' do
    infer 'A.1.'
    infer 'A.1.2'
  end
  infer 'A.2'
end
```

On agit de manière récursive. Pour créer l'arbre de `A`, on crée tous ses arbres après avoir incrémenté `@depth`. Ainsi, ses sous-arbres seront stockés dans `@children[@depth + 1]`. Une fois que ses sous-arbres ont été créés (et sont dans `@children[@depth + 1]`), on crée un nouvel arbre dont les prémisses sont ses sous-arbres (donc `@children[@depth + 1]`). Ensuite, on vide `@children[@depth + 1]` (le potentiel `infer` de la même preuve doit repartir de zéro pour ses sous-arbres) et on ajoute l'arbre qu'on vient de créer à `@depth[tree]` (les arbres de sa profondeur).

Un déroulement à la main sur un exemple aidera à bien comprendre son fonctionnement.

3. Le passage à LaTeX

Maintenant que nous pouvons écrire nos preuves, il est temps de les transformer en LaTeX à l'aide de la méthode `format` de `ProofTree`. Ici, nous allons le faire pour `ebproof`. Ce n'est pas très compliqué. On formate les prémisses, et ensuite on formate le contenu du nœud courant. La commande à utiliser est `\hypo<nb_premises>`.

Néanmoins, histoire de ne pas avoir toute la preuve LaTeX en un seul bloc, on va également indenter le code LaTeX, à chaque fois qu'on va dans un sous-arbre, on indente un peu plus. Tout ceci peut mener à ce code.

```ruby
class EBProofFormatter
  INDENT = "  "

  def initialize(indent: INDENT)
    @indent = indent
  end

  def format(tree)
    content = format_body(tree)
    "\\begin{prooftree}\n#{content}\\end{prooftree}"
  end

  private

  def format_command(tree)
    "\\infer#{tree.premises.size}"
```

```
  end

  def format_label(tree)
    "[$#{tree.label.latex}$]"
  end

  def format_sequent(tree)
    command = format_command(tree)
    label = format_label(tree)
    "#{command}#{label}{#{tree.conclusion}}"
    end
  end

  def format_body(tree, depth: 0)
    indent = @indent * depth
    content = format_sequent(tree)
    precedent = tree.premises.map { format_body(_1, depth: depth +
        1) }.join
    "#{precedent}#{indent}#{content}\n"
  end
end
```

Et on peut maintenant écrire ce code.

```
contradictory_context = Proof.new do
  infer 'A, \neg A \vdash B', with: ElimFalse do
    infer 'A, \neg A \vdash \bot ', with: ElimImpl do
      infer 'A, \neg A \vdash A', with: Axiom
      infer 'A, \neg A \vdash \neg A', with: Axiom
    end
  end
end

formatter = EBProofFormatter.new
puts contradictory_context.tree.format(formatter)
```

Qui nous donne ce code (qui est le code LaTeX que nous avons vu en introduction).

```
\begin{prooftree}
      \infer0[$\text{ax}$]{A, \neg A \vdash A}
      \infer0[$\text{ax}$]{A, \neg A \vdash \neg A}
   \infer2[$\implies_\text{e}$]{A, \neg A \vdash \bot}
\infer1[$\bot_\text{e}$]{A, \neg A \vdash B}
\end{prooftree}
```

4. Une meilleure syntaxe

Pour être honnête, la syntaxe que nous avons pour écrire nos preuves est peut-être un peu mieux sur certains points, mais sur d'autres, c'est clairement pas fou. Par exemple, on devra dans certains cas échapper les `\` (dans notre exemple on s'en sort bien en utilisant des guillemets simples), et quand on doit écrire des commandes LaTeX, c'est pas le plus simple.

Ce qu'il nous faut, c'est permettre à l'utilisateur de définir des symboles qui seront ensuite remplacés par du code LaTeX! Par exemple, nous pourrions écrire la méthode précédente de cette manière.

```
new_contradictory_context = Proof.new do
  infer "A, !A => B", with: ElimFalse do
    infer "A, !A => Fa ", with: ElimImpl do
      infer "A, !A => A", with: Axiom
      infer "A, !A => A -> Fa ", with: Axiom
    end
  end
end
```

Là, on a quelque chose qui se lit et s'écrit plutôt facilement! En plus, ce n'est pas très compliqué à implémenter si nous nous limitons à du simple remplacement (dans notre exemple, `!` sera remplacé par `\neg`, `=>` par `\vdash` et `Fa` par `\bot`). Nous créons une classe `Texifier` qui permet de transformer une chaîne de caractère en appliquant des remplacements.

```
class Texifier
  def initialize(correspondances)
    @correspondances = correspondances.to_h
  end

  def texify(text)
    @correspondances.reduce(text) do |str, (symbol, command)|
      str.gsub(symbol, command)
    end
  end
end
```

Par exemple, voici une instance de `Texifier` qui peut être utilisée pour la logique.

```
BASIC_TEXTIFIER = Texifier.new([
  ["<-> ", "\\iff "],
  ["<->", "\\iff"],
  ["-> ", "\\implies "],
  ["->", "\\implies"],
  ["/\\ ", "\\land "],
```

```
  ["/\\", "\\land"],
  ["\\/ ", "\\lor "],
  ["\\/", "\\lor"],
  ["=> ", "\\vdash "],
  ["=>", "\\vdash"],
  ["Exists ", "\\exists "],
  ["Forall ", "\\forall "],
  ["Fa ", "\\bot "],
  ["Tr ", "\\top "],
  ["!", "\\neg "]
].freeze)
```

Notre `Texifier` va alors être utilisé dans certaines classes de formatage. Nous rajoutons un paramètre `formatter` à `EBProofFormatter` et une variable d'instance `@formatter` qui sera utilisée dans `format` pour *texifier* le contenu du nœud.

```ruby
# Le nouveau constructeur de EBProofFormatter
def initialize(indent: INDENT, texifier: BASIC_TEXTIFIER)
  @indent = indent
  @texifier = texifier
end

# La nouvelle méthode format de EBProofFormatter
def format(tree)
  content = format_body(tree)

      "\\begin{prooftree}\n#{@texifier.texify(content)}\\end{prooftree}"
end
```

Et on peut tester nos nouveautés.

```ruby
formatter = EBProofFormatter.new(texifier: BASIC_TEXIFIER)
puts new_contradictory_context.tree.format(formatter)
```

Qui nous donne bien le résultat attendu, les symboles ont été remplacés par leur équivalent LaTeX!

Finalement, on a bien réussi à obtenir un outil pour écrire des preuves avec une syntaxe plus simple. Si on voulait faire un petit jeu de mots, on dirait que c'est une preuve de concept! Il reste cependant des choses que je n'ai pas présentées ici mais que j'ai dans mon code (qui diffère d'ailleurs de celui écrit dans ce billet).

— La possibilité de ne pas avoir de ligne au-dessus d'une conclusion sans prémisses comme dans l'exemple de gauche ci-dessous.

$$\dfrac{\dfrac{A, \neg A \vdash A \qquad A, \neg A \vdash A \implies \bot}{A, \neg A \vdash \bot} \implies_{\mathrm{e}}}{A, \neg A \vdash B} \bot_{\mathrm{e}} \qquad \text{contre} \qquad \dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}\mathrm{ax} \qquad \dfrac{}{A, \neg A \vdash A \implies \bot}\mathrm{ax}}{A, \neg A \vdash \bot} \implies_{\mathrm{e}}}{A, \neg A \vdash B} \bot_{\mathrm{e}}$$

— La possibilité d'avoir des points verticaux (avec un label à droite), par exemple pour représenter une induction (voir la commande `\ellipsis` de ebproof).

J'ai actuellement créé une gem pour ce projet. Elle n'est pas publiée pour le moment, mais le sera certainement à un moment. De plus, je travaille également sur un projet un peu plus gros pour écrire du LaTeX de manière générale à l'aide de Ruby, et pas seulement des arbres de preuves.

Liste des abréviations

DSL Domain Specific Language. 1, 3, 4