Pieuvre: le mini assitant de preuve

Marwan et Augustin

1 Présentation

Nous avons programmé **Pieuvre**, un assistant de preuve pour la logique intuitionniste en **OCaml**. Il permet, à l'aide d'une boucle interactive, de prouver des prédicats simples.

Nous indiquons dans la partie 2 comment compiler, lancer et utiliser notre programme. Nous exposons ensuite à la partie 3 comment notre programme est structuré. Nous présentons enfin dans la partie 4 notre implémentation du λ-calcul simplement typé puis dans les parties 5 et 6 notre implémentation des preuves et des tactiques.

2 Compiler et exécuter

Pour compiler : `make pieuvre`

Pour exécuter **Pieuvre** et entrer dans la boucle interactive de l'assistant de preuve, il suffit d'entrer la commande `./pieuvre`. Sinon, on peut spécifier un des modes suivants :

— `-alpha`
  qui vérifie l'α-équivalence de deux λ-termes séparés par `&`.

— `-reduce`
  qui affiche les β-réduction successives du λ-terme.

3 Organisation du code

Le code est structuré de la manière suivante :

— `parser.mly` et `lexel.mll` qui définissent l'analyse syntaxique du λ-calcul simplément typé ainsi que des tactiques.

— `lam.ml` qui contient notre implémentation du λ-calcul simplement typé.

— `types.ml` qui contient la définition des `types` et `typing.ml` qui implémente l'algorithme de vérification de type.

— `hlam.ml` qui décrit les λ-termes creux, utilisés dans les preuves.

— `proof.ml` qui définit la structure d'une preuve et implémente les tactiques.

— `main.ml` le point d'entrée de notre programme qui fait l'interface avec l'utilisateur.

4 λ-calcul simplement typé

Avant de pouvoir réaliser des preuves, il faut pouvoir les encoder. On utilise le λ-calcul simplement typé. Dans notre programme, on utilise la syntaxe suivante pour les types et pour les termes :

```
⟨A, B⟩ ::= X
  | A -> B
  | A /\ B
  | A \/ B
  | false

⟨M, N⟩ ::= x
  | fun (x : A) => M
  | M N
  | exf (M : A)
  | l(M : A \/ B)
  | r(M : A \/ B)
  | (M, N)
```

Les termes ainsi définis se réduisent à l'aide des règles de β-réduction rappelées dans les notes de Selinger [3]. Avant d'être β-réduit par `betastep`, les termes sont d'abord α-convertis avec `alpha_convert` afin d'éviter les problèmes de substitution.

On remarque que l'on impose d'annoter le type de l'argument d'une λ-abstraction, des constructeurs de termes somme et produit. On peut alors vérifier le type de tous les termes de manière déterministe avec les règles rappelées dans le livre de S. Mimram [2] et *très* brièvement dans le livre de Landin [1].

L'implémentation du OU et du ET est incomplète. Nous n'avons pas eu le temps d'implémenter les déconstructeurs associés (les projecteurs et le `match`).

5 Preuves

Pour construire la preuve d'une proposition, on se base sur la correspondance de Curry-Howard : ainsi, construire une preuve correspond à créer un λ-terme dont le type correspond à la proposition.

Les λ-termes de preuves sont des λ-termes particuliers au sens où ils sont **creux**. Intuitivement, les λ-termes creux, ou `hlam`, sont des termes avec des trous. Compléter la preuve correspond à remplir ces trous à l'aide de tactique.

Lorsqu'on déclare un but, on lui assigne donc :

— un lambda terme creux (`Ref (ref Hole)`) que l'on construit petit à petit à l'aide des tactiques.

— le type associé à ce que l'on souhaite prouver

— son contexte, c'est à dire un liste d'hypothèses (nom de l'hypothèse, type, λ-terme possiblement encore creux qui sera mis à jour facilement grâce aux `ref`s)

Au moment du `Qed`, ce λ-terme de preuve qui a été gardé de côté est β-réduit puis typé contre le type du but définit au moment de `Goal`.

6 Tactiques

Dans le mode générique, le programme démarre un assistant de preuve interactif semblable à Coq. Voici les commandes et tactiques implémentées :

- `Goal` pour émettre un nouveau but
- `Undo` pour annuler la dernière tactique. Son implémentation est très brutale, on se contente de maintenir une pile des états du terme de preuve. Appliquer cette tactique revient à dépiler l'état courant et à revenir au précédent. L'implémentation pourrait être plus judicieuse.
- `Qed` pour clore la preuve d'un but
- `Check` construit le fichier `checker.v` qui contient le but courant ainsi qu'un `exact` du terme de preuve. Ce fichier est envoyé au top-level de Coq afin d'être vérifié.
- `exact` si une hypothèse correspond parfaitement à notre but courant
- `assumption` recherche automatique d'une hypothèse correspondant au but courant
- `intro` pour défaire une implication
- `intros` pour répéter `intro` autant que possible
- `cut` permet de faire une élimination de la coupure.
- `apply H` où `H` est une implication avec le but courant en conclusion permet de prouver directement les prémisses de l'implication.
- `split` permet de prouver un côté puis l'autre d'une conjonction
- `left` et `right` permettent de choisir quel côté de la conjonction prouver
- `try` pour essayer une tactique. Si elle échoue, rien ne se passe.

Là encore, il manque la tactique `destruct`.

On peut prouver `A -> A /\ A` mais pas l'implication inverse.

7 Conlusion

Nous avons présenté notre assistant de preuve **Pieuvre**. Il permet de prouver des prédicats logiques simples. Plusieurs extensions (et améliorations) sont envisageables.

Tout d'abord, on pourrait évidemment enrichir la grammaire et les tactiques pour avoir un langage plus expressif. On pourrait également laisser la possibilité à l'utilisateur de prouver des `lemmes` et des `théorèmes` pour les sauvegarder et les réutiliser plus tard. Il faudrait alors également proposer les types dépendants.

Enfin, il faut se poser la question de la fiabilité de notre assistant de preuve. Il serait difficile de prouver formellement toutes les composantes du programme (la β-réduction, l' α-conversion, le `typecheck`, etc.). Ainsi, on propose de sauvegarder les termes de preuves puis de les envoyer à Coq pour valider la preuve. Cette approche est assez satisfaisante dans la mesure où la construction du terme

de preuve repose uniquement sur les tactiques et pas sur notre implémentation du λ-calcul. Ainsi, on peut supposer qu'on n'a jamais de faux-positif : si Check valide notre preuve, c'est quelle est valide.

8 Répartition du travail

— parser et lexer pour le λ-calcul et les tactiques (Marwan)
— exfalso et typage bidirectionnel (Marwan)
— β-réduction (Marwan)
— système de preuve (définition des `hlam`, des contextes, des buts, etc.) (Marwan)
— les tactiques exact, assumption, intro, intros, cut, apply et try (Marwan)
— α-équivalence / renommage (Augustin)
— fonctionnement du `main` (options, arguments, etc.) (Augustin)
— boucle interactive et interface utilisateur (Augustin)
— les tactiques Goal, Qed, left, right et split (Augustin)
— Undo (Augustin)

Références

[1] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3), 1966.

[2] Samuel Mimram. *PROGRAM = PROOF*. 2020.

[3] Peter Selinger. Lecture notes on the lambda calculus. *CoRR*, 2008.