Implementing a propositional prover

Thomas FOURIER

15 décembre 2023

1 Inférence de types pour du calcul simplement typé

Les types sont définis de manière assez naturelle, avec des notations préfixes. Les notations infixes pour l'implication, le /\ et \/ auraient été plus naturelles, mais OCaml ne semble pas supporter cela dans la définition des types.

Pour ce qui est des types, la plupart sont intuitifs. J'ai introduit les deux projections `Fst Snd` pour la conjonction. Pour faciliter la disjonction, j'ai ajouté le type de l'autre membre. Ce n'est pas nécessaire, on aurait pu inférer le type du membre dont on n'a pas l'expression en analysant l'ensemble de la formule pour déduire quels types doivent être égaux. Par exemple, dans :

```
( "x",
    Or (TVar "A", TVar "B"),
    Case
        ( Var "x",
        Abs ("a", TVar "A", Right (Var "a")),
        Abs ("b", TVar "B", Left (Var "b")) ) )
```

on sait que les deux expressions `Right (Var "a")` et `Left (Var "b")` ont le même type. On peut donc en déduire que le type est `Or (TVar "A", TVar "B")`. Si on ne peut pas déduire le type d'un des deux membres, c'est qu'il n'a pas été utilisé, on peut donc lui donner un nom unique arbitraire. Je n'ai pas implémenté cette fonctionnalité parce que cela aurait demandé beaucoup de modifications et avoir une structure de type particulier.

Le `Case` attend des arguments de types particuliers. Le premier est un `Or(A, B)`, le deuxième `Imp(A,C)`, `Imp(B,C)`. `Case(x, f, g)` représente, si `x = Left(y, _)` `f(x)` et si `x = Left(_, y)` `f(y)`. Cette approche est différente du cours où on avait un constructeur `Case` qui prend cinq arguments. Les deux sont équivalents. J'ai choisi ma solution parce qu'elle permet d'utiliser le constructeur `App` et de ne pas avoir à le recoder.

Le constructeur `Rec` prend trois arguments : le premier représente un entier, le second le cas de base (c'est-à-dire la valeur si l'entier est nul, de type `A`) et une fonction de type `Imp(Nat, Imp(A, A))`. La sémantique est :

```
Imp(Z, init, _) = init
Imp(S n, init, hered) = hered n (Imp(n, init, hered))
```

La fonction `string_of_tm` n'est pas très claire pour ce constructeur, mais je n'ai pas trouvé une manière plus pertinente de l'afficher. Par exemple, la somme s'écrit :

```
(fun n:Nat) -> Match n with
  |0 -> (fun m:Nat) -> m
  |succ n ->  with n+ = fct evaluated at pred -> (fun m:Nat) -> Suc of (n+ m)
```

Pour faciliter la lecture de certaines fonctions, j'ai créé une fonction `nat_to_int` qui permet d'afficher les entiers de manière naturelle. Cela servira plus dans la partie 5.

Pour l'inférence de types, on vérifie pour `App`, `Case` et `Rec` que les types sont bien valides comme définit plus haut.

Pour le démonstrateur interactif, l'introduction permet de traiter l'implication et la conjonction séparément. On traite les deux cas de manière naturelle : pour `A => B`, on suppose `A` et on prouve `B`, ce qui revient à se donner une variable de type `A` et calculer un résultat de type `B`. Pour la conjonction, on montre d'abord le terme de droite puis celui de gauche. J'ai noté que l'expression `let ... and ... in` n'est pas exécuté dans l'ordre.

La commande `exact` est plus simple, il suffit de vérifier que la variable a le type à démontrer.

Les méthodes `elim` sont plus diverses. Il y a plusieurs méthodes pour l'élimination du `Nat`. Le mot clé `elim` permet de faire une récurrence. Avec les notations utilisées pour introduire la récurrence, on doit d'abord montrer `A`, ce qui correspond à la valeur en `0`. On doit ensuite définir l'hérédité. Pour cela, on utilise l'entier sur lequel on fait l'élimination comme premier paramètre et on doit prouver `A => A`.

Les deux autres méthodes d'élimination des entiers : zéro et successeur sont des mots clés dédiés. Successeur ne change rien au contexte, ni à la formule à prouver si elle est licite (c'est-à-dire égale à `Nat`). Le zéro prouve `Nat`.

2 Dependant types

Pour la substitution, j'ai utilisé l'idée du TD précédent, c'est-à-dire que je ne remplace pas le nom de la variable dans les abstractions et dans les Π si la variable est déjà utilisée. Cela évite de remplacer tous les noms des variables. Cela permet de conserver les noms de variables définies à la main s'il n'y a pas de collision.

La normalisation utilise une fonction auxiliaire : `red_aux`. Cette fonction retourne une paire `expr, bool` qui correspond à l'expression réduite et un booléen vrai si l'expression a été réduite. La normalisation consiste à réduire jusqu'à ce qu'aucune réduction ne soit possible. Pour une expression bien typée, la normalisation termine.

Les réductions difficiles sont l'application, l'abstraction et l'induction. Pour l'implication, je réduis chaque terme, et si je peux appliquer directement parce que le premier terme est une abstraction, je le fais. À cette étape, je ne vérifie pas les types. Si les expressions sont typables, l'application est licite. Pour l'abstraction, on doit indiquer dans l'environnement le type de la variable au moment de réduire le terme de droite. La réduction du `Pi` s'effectue de la même manière. Pour l'induction, je commence par réduire au maximum chacun des termes. Lorsque chaque terme est réduit je peux commencer à appliquer l'induction, c'est-à-dire renvoyer l'initialisation si on applique l'induction à zéro et appliquer l'hérédité pour le successeur d'un entier. J'ai choisi de réduire les termes d'abord pour ne pas avoir à le faire plusieurs fois après avoir commencé à appliquer l'hérédité.

La fonction `alpha` vérifie que les constructeurs sont les mêmes et que chaque terme est α-équivalent. Lorsque l'on introduit des variables (avec un `Abs` ou un `Pi`), je remplace l'un des noms de variable par l'autre et je continue récursivement.

C'est au moment de l'inférence de type que l'on doit vérifier que toutes les constructions sont licites. Pour l'application, je vérifie que le terme de gauche est bien une fonction qui peut

s'appliquer au terme de droite.

Pour l'induction, on doit vérifier que le type est valable pour toutes les valeurs de n. Pour cela, on raisonne par induction : si n vaut 0, on utilise `typ` pour connaître le type attendu et on vérifie que l'initialisation est bien de ce type (lignes 235–236). Pour la récurrence, on vérifie qu'en appliquant l'hérédité, on obtient bien le même type qu'en appliquant `typ` au successeur du nombre.

J'ai défini, comme dans la première partie une fonction qui tente de transformer une expression de type `Nat` en un entier. Cela permet d'améliorer l'affichage. Par ailleurs, on peut entrer des entiers dans l'interface, par exemple, en entrant `N2`, on obtient `S (S Z)`.