Preuves assistées par ordinateur

Examen du 15 mai 2023 – durée : 3h – avec typos corrigées

Documents autorisés

Le barème ci-dessous est indicatif et est susceptible d'être modifié.

Exercice 1 (≈ 6 points) – Déduction naturelle

1. Les séquents suivants sont-ils dérivables en déduction naturelle ? Si oui, en donner des dérivations en forme normale (au choix : soit en style arbre de formules, soit en style arbre de séquents, soit sous la forme d'un λ-terme de preuve). Si non, donner un contre-modèle (au choix : en assignant aux atomes un booléen qui rend le séquent faux ou en montrant qu'aucune forme normale n'est possible).
   (a) $(A \wedge B) \wedge C \ \vdash \ (B \wedge C) \wedge A$
   (b) $(A \Rightarrow B \Rightarrow C) \ \vdash \ (A \wedge B) \Rightarrow C$
   (c) $(\exists x\, A(x)) \Rightarrow B \ \vdash \ \forall x\, (A(x) \Rightarrow B)$
   (d) $((A \wedge B) \vee C) \ \vdash \ A \wedge C$
   (e) $(A \vee B) \wedge C \ \vdash \ A \vee (B \wedge C)$
2. On se place dans l'arithmétique où les entiers sont construits à partir de 0 (zéro) et S (successeur). On écrit 1 pour (S 0).
   On considère la preuve suivante par analyse de cas de $\forall x \exists y\, y \leq S\, x$:

   $$H := \lambda x.\texttt{match}\ x\ \texttt{with}\ 0 \Rightarrow (1, \mathit{refl}\ 1) \mid S\ n \Rightarrow (n, ax\, n)\ \texttt{end}$$

   où *refl* est une preuve préexistante de $\forall x\, x \leq x$ et où ax est une preuve préexistante de $\forall x\, x \leq S\, (S\, x)$.
   Donner la forme normale de la preuve $H\, 1$ de $\exists y\, (y \geq 1)$ ainsi que la forme normale de la preuve $H\, 0$ de $\exists y\, (y \geq 0)$ (on pourra si nécessaire faire référence à *refl*, ax).

Exercice 2 (≈ 4 pts) : Iteration, analyse de cas et récursion

On considère le type suivant de suites de booléens, défini en Coq par :

```
Inductive bool_list :=
| Nil : bool_list
| Cons_true : bool_list -> bool_list
| Cons_false : bool_list -> bool_list.
```

On suppose ce type équipé d'un schéma de récursion dépendante :

```
bool_list_rect : forall B:bool_list -> Type, B Nil ->
            (forall l:bool_list, B l -> B (Cons_true l)) ->
            (forall l:bool_list, B l -> B (Cons_false l)) ->
             forall l:bool_list, B l
```

Pour chacune des questions suivantes, on donnera la preuve sous la forme d'un λ-terme, en syntaxe informelle ou syntaxe Coq.

1. Indiquer comment prouver à partir de `bool_list_rect` un schéma de récursion non dépendante `bool_list_rect_no_dep` de type :

```
bool_list_rect_no_dep : forall B:Type, B ->
                   (bool_list -> B -> B) ->
                   (bool_list -> B -> B) ->
                    bool_list -> B
```

2. Indiquer comment prouver à partir de `bool_list_rec_no_dep` le schéma d'analyse de cas `bool_list_case_no_dep` de type suivant :

```
bool_list_case_no_dep : forall B:Type, B ->
                   (bool_list -> B) ->
                   (bool_list -> B) ->
                    bool_list -> B
```

3. Indiquer comment prouver à partir de `bool_list_rec_no_dep` ls schéma d'itération `bool_list_iter` de type suivant :

```
bool_list_iter : forall B:Type, B ->
           (B -> B) ->
           (B -> B) ->
            bool_list -> B
```

Exercice 3 ($\approx$ 16 pts) : Simulation des exceptions dans un langage purement fonctionnel

Notez que certaines questions peuvent être traitées indépendamment.

A – Langage d'expressions arithmétiques

On considère le type suivant `expr` décrivant en Coq un langage d'expressions arithmétiques :

```
Inductive expr :=
| Constant : nat -> expr      (* une constante entière *)
| Mul : expr -> expr -> expr  (* le produit formel de deux expressions *)
| IfZero : expr -> expr -> expr -> expr (* évalue la 2e expr si 1e égale 0, la 3e sinon *).
```

1. Écrire en Coq une fonction de type `expr -> nat` qui évalue l'expression donnée en entrée.

B – Langage d'expressions arithmétiques avec possible division par 0

On étend maintenant notre langage avec la division euclidienne :

```
Module Implicit.
Inductive expr :=
| Constant : nat -> expr
| Mul : expr -> expr -> expr
| IfZero : expr -> expr -> expr -> expr
| Div : expr -> expr -> expr  (* la division de deux expressions *).
End Implicit.
```

Comme la division par 0 n'est pas définie, l'évaluation d'une expression n'est plus forcément un entier car une division par zéro peut avoir eu lieu.

Dans une première approche, comme il n'y a pas de mécanisme d'exception en Coq, on va simular la présence possible d'une exception dans un type `option` et considérer un évaluateur de type `expr -> option nat` qui va renvoyer `Some` d'un entier quand l'évaluation réussit et `None` sinon.

2. Écrire en Coq, dans le module `Implicit`, une fonction `interp` de type `expr -> option nat` qui évalue l'expression donnée en entrée et l'encapsule dans un `Some`, à moins qu'une division par zéro n'ait eu lieu, auquelle cas l'erreur est signalée en retournant `None` (notez qu'une exception peut arriver à n'importe quel moment de l'évaluation et qu'il va donc falloir traiter le cas d'une possible exception dans tous les appels récursifs).

   Si on le souhaite, on pourra utiliser l'une ou l'autre des fonctions auxiliaires suivantes :

```
Definition bind {A B : Type} (a : option A) (f : A -> option B) : option B :=
  match a with
  | None => None
  | Some a => f a
  end.

Definition ret {A : Type} (a : A) : option A := Some a.
```

C – Langage d'expressions arithmétiques avec primitives de gestion des exceptions

Dans une deuxième approche, on veut pouvoir utiliser un évaluateur standard quitte à changer le langage d'expression. Ainsi, on ajoute explicitement dans le langage des nouvelles constructions pour gérer les exceptions. Comme toutes les expressions du langage n'ont plus alors le même type, on ajoute un type aux expressions :

```
Module Explicit.
Inductive expr : Type -> Type :=
| Constant : nat -> expr nat
| Mul : expr nat -> expr nat -> expr nat
| IfZero : expr nat -> expr (option nat) -> expr (option nat) -> expr (option nat)
| SafeDiv : expr nat -> expr nat -> expr nat
  (* une variante de division qui n'échoue pas si le diviseur est 0 *)
  (* (elle renvoie alors un résultat arbitraire) *)
| DivByZero : expr (option nat)
| Return : expr nat -> expr (option nat)
  (* pour injecter un résultat normal comm résultat potentiellement avec erreur *)
| Bind : expr (option nat) -> (nat -> expr (option nat)) -> expr (option nat)
  (* évalue le 1er argument et propage l'erreur potentielle, *)
  (* si pas d'erreur, applique le résultat au 2e argument *).
```

3. Écrire, dans le module `Explicit`, un nouvel interpréteur `interp`, qui a cette fois le type `forall T, expr T -> T` sachant que l'évaluation d'une expression de type `expr T` est destinée à être évaluée en `T`.
4. Écrire aussi une fonction de traduction `trad` qui transforme une expression du module `Implicit` en une expression du module `Explicit`. Son type est donc :

```
trad : Implicit.expr -> Explicit.expr (option nat)
```

Par exemple, la traduction de l'expression `Mul E1 E2` sera

```
Bind E1' (fun n1 => Bind E2' (fun n2 => Return (Mul (Constant n1) (Constant n2))))
```

où `E1'` et `E2'` sont les traductions de `E1` et `E2`. La signification est d'évaluer `E1'`, de propager l'erreur si l'évaluation a produit une erreur, de sinon évaluer `E2'`, de propager l'erreur si cela a produit une erreur, et de sinon retourner la multiplication des deux valeurs en les injectant dans le type `option nat`.

La traduction de l'expression `Div E1 E2` sera elle

```
Bind E1' (fun n1 =>
Bind E2' (fun n2 =>
IfZero a2 DivByZero (Return (SafeDiv (Constant n1) (Constant n2)))))
```

La traduction peut paraître compliquée mais elle est en fait très uniforme. On appelle cela une *traduction monadique* et il y a une théorie générale des traductions monadiques qui permet de simuler tout effet de bord (exceptions, affectations, goto, random, ...) dans un langage purement fonctionnel (sans effets de bord). Cette théorie est abondamment utilisée dans le langage Haskell.

5. Si tout est fait correctement, on peut prouver le théorème suivant :

```
Theorem correction :
   forall E:Implicit.expr, Implicit.interp E = Explicit.interp (trad E).
```

Indiquer les étapes clés de la preuve.

Maintenant que l'architecture est en place pour manipuler les exceptions, on va rajouter un rattrapeur d'exception.

```
Inductive expr : Type -> Type :=
| Constant : nat -> expr nat
| Mul : expr nat -> expr nat -> expr nat
| IfZero : expr nat -> expr (option nat) -> expr (option nat) -> expr (option nat)
| SafeDiv : expr nat -> expr nat -> expr nat
| DivByZero : expr (option nat)
| Return : expr nat -> expr (option nat)
| Bind : expr (option nat) -> (expr nat -> expr (option nat)) -> expr (option nat)
| CatchDivByZero : expr (option nat) -> expr nat -> expr nat
  (* retourne la valeur du 1er argument si pas erreur, sinon évalue le 2e *).
```

6. Étendre la fonction `Explicit.interp` avec le cas `CatchDivByZero`.

D – Théorie générale de la traduction monadique des effets de bord

Plus généralement, la théorie des traduction monadiques repose sur :
— une transformation de type, appelée monade, ici `T := option`
— un "return" qui dit comment injecter un résultat normal dans la monade, ici
`Definition ret {A} (a : A) : option A := Some a.`
— un "bind" qui dit comment enchaîner des résultats pouvant faire des effets de bord, ici, donc,

```
Definition bind {A B : Type} (a : option A) (f : A -> option B) : option B :=
  match a with None => None | Some a => f a end.
```

Avec trois propriétés :

```
Theorem bind_return : forall {A B} (t : A) (f : A -> T B), bind (ret t) f = f t.
Theorem return_bind : forall {A B} (t : T A) (f : A -> T B), bind t ret = t.
Theorem bind_bind : forall {A B C} (t : T A) (f : A -> T B) (g : B -> T C),
  bind (bind t f) g = bind t (fun a => bind (f a) g).
```

7. Montrer informellement pourquoi ces trois théorèmes sont effectivement vrais dans le cas du `return` et du `bind` de la transformation `option`.

Une autre monade très courante est la monade d'état sur une mémoire globale `m` de type `B`, définie par `State A := B -> A * B`.

Par exemple, le `bind` pour cette monade est défini par :

```
Definition bind {A B : Type} (a : State A) (f : A -> State B) : State B :=
  fun b => let (a',b') := a b in f a' b'.
```

8. Avez-vous une idée de ce que pourrait être la définition du `return` pour la monade d'état ? En tout cas, il doit avoir le type `A -> State A`, c'est-à-dire `A -> B -> A * B`. Facile !
9. De la même manière que l'intérêt de la monade option est de pouvoir y interpréter une exception et un rattrapeur d'exception, l'intérêt de la monade d'état est de pouvoir y interpréter l'écriture et la lecture dans la mémoire `m` de type `B`.

   L'écriture a le type `B -> State unit` et peut se définir par

   ```
   Definition write (b:B) : State unit := fun b => (tt,b).
   ```

   où `tt` est l'unique habitant de `unit`.

   Avez-vous une idée de comment définir la lecture `read` qui doit avoir le type `unit -> State B`, c'est-à-dire `unit -> B -> B * B` ? Facile !

Annexe

On rappelle la définition des types `option` et `unit` en Coq :

```
Inductive option (A:Type) : Type :=
| Some : A -> option A
| None : option A.

Inductive unit : Type := tt.
```

On rappelle aussi que la multiplication et la division euclidienne sur les entiers s'appellent respectivement `Nat.mul` et `Nat.div`.