Vérification

Groupe D

October 17, 2023

1 Introduction

Il nous a été rappelé qu'une introduction au lambda calcul pourrait nous être nécessaire pour mieux appréhender les concepts relatif à coq , j'aime bien partir de la page wikipédia sur le lambda calculus [?] pour délimiter les thèmes , puis lire attentitvement les 50 premières pages de l'introduction au lambda calculus présenté par Boro Sitnikovski [8] et finalement le chapitre 3 et plus particulièrement 5 du Coq Art **??**. Pour avoir une ressource "plus académique" sur le calcul des constructions se référer au cours de Mme.Paulin **??** .

2 Le langage :

Tout d'abord nous allons définir un langage sur lequel nous allons travailler (l'écrire "formellement" nous sera très utile pour la suite de ce TP) .

2.1 Qu'est ce qu'un langage ?

Syntaxe :

Un langage est la réunion d' ensemble de symboles chacun exprimant : les variables , les constantes , les fonctions et les relations

Sémantique :

Un langage L (une signature) est interprété par une L-structure (une interprétation) qui associe une valeur sémantique à chaque symbole de L . Cette structure est définit sur B (le domaine/la base d'interprétation) .

2.2 De quoi avons nous besoin ?

{while ,assignation , ... } $\xRightarrow{Stm}$ les instructions

Une instruction while est de la forme:

```
while ( Bexp:cond ) { Stm:programme }
```

Donc nous avons besoin d'une meta variable pour exprimer la condition ainsi que d'une sémantique pour pouvoir l'évaluer .

Une assignation est de la forme :

```
Ass(A:Type , lhs:Var_A , rhs:Expression_A)
// A est le type de la variable lhs et rhs est une expression convertible en ce type
```

Donc nous avons besoin de meta variables pour exprimer les variables et les expressions de type A .

Dans le cadre de ce TP nous nous restreindrons aux expressions arithémtiques définit sur $\mathbb{Z}$ donc A sera toujours égal Z .

2.3 Traduction formelle :

Nous présupposons que la structure associé aux entiers relatifs est déjà définit donc nous pouvons l'utilisé dans nos constructions futures .

2.3.1 Expression Arithmétique :

Syntaxe:
Soit a le symbole de variables .
Soit $\{:=, -, +, \div, \times\}$ les symboles de relations.

Sémantique:
Soit $\mathbb{Z}$ la base de la L-structure
Soit $\{Ass, Minus, Plus, Div, Mult\}$ les interprétation des relations dans $\mathbb{Z}$.

2.3.2 Expression Booléenne :

Syntaxe:
Soit b le symbole de variables .
Soit $\{\neg, \wedge, \implies, =, <=\}$ les symboles de relations
Soit $\{BTrue, BFalse\}$ les "symboles" de constantes .

Sémantique:

Soit $\mathbb{B}$ la base de la L-structure
Soit $\{Not, And, Eq, Le\}$ les interprétation des relations dans $\mathbb{B}$.
Soit $\{true, false\}$ l'interprétations des constantes dans $\mathbb{B}$.

2.4 Implémentation Coq :

2.4.1 Rappel Type inductif :

Un type inductif est construit par un ensemble de constructeurs

Inductive nom $(p^1 : P1)...(p^{np} : P^{np})$:$\forall(a^1 : A^1)...(a^{na} : A^{na})$, s :=
co^1 :$\forall(x_1^1 : A_1^1)...(x_1^{a1} : A_1^{a1}), nom(p^1, ..., p^{np}, t^1, ..., t^{na})$
...
co^n :$\forall(x_n^1 : A_n^1)...(x_n^{an} : A_n^{an}), nom(p^1, ..., p^{np}, t^1, ..., t^{na})$

Vocabulaire :

- $\{p^1, ..., p^n\}$ sont appelés les paramètres de la famille car l'ensemble des constructeurs est une famille qui nécessite ces paramètres .
- s est la sorte de nom , la sorte s peut avoir besoin de paramètre pour se définir d'où la présence des $\{a^1, ..., a^n\}$ (cf. type dependent) .
- $\{a^1, ..., a^n\}$ sont appelé les paramètres de prédicat , l'explication est qu'un type dependent peut s'écrire sous la forme d'un prédicat , un prédicat s'écrit :
  $A^1 \rightarrow ... \rightarrow A^n \rightarrow Prop$.

Nous remarquons que chaque constructeur prend un certain nombre d'arguments de type divers et renvoie une expression de la forme :

$$\boldsymbol{nom(p^1, ..., p^{np}, t^1, ..., t^{na})}.$$

Cela veut dire que chacun des n constructeurs de nom peut construire une expression de type :$nom(p^1, ..., p^{np}, t^1, ..., t^{na})$ à partir d'une autre expression définit avec ai arguments ($i \in [n]$) .

Pour l'instant l'on ne s'interesse qu'à l'interprétation/évaluation des symbole de relation et de fonction . Nous verrons plus en détail l'interprétations des symboles de variables et constante dans une autre partie .

2.4.2 Expression Arithmétique :

Syntaxe: La syntaxe est donnée par le type inductif suivante :

```
Inductive aexp : Type :=
| ANum : Z -> aexp
| AVar : ... -> aexp //(* on le completera plus tard *)
| APlus : aexp -> aexp -> aexp
| AMinus : aexp -> aexp -> aexp
| AMult : aexp -> aexp -> aexp.
```

Sémantique: La sémantique est donnée par la fonction d'évaluation suivante :

```
Fixpoint aeval (a : aexp) : Z :=
  match a with
    | ANum n => n //(* on le verra plus tard *)
    | AVar v => ... //(* on le completera plus tard *)
    | APlus a1 a2 => (aeval a1) + (aeval a2)
    | AMinus a1 a2 => (aeval a1 ) - (aeval a2)
    | AMult a1 a2 => (aeval a1) * (aeval a2)
  end.

```

2.4.3 Expression Booléenne :

Syntaxe:

```
Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.
```

Sémantique:

```
Fixpoint beval (e : bexp) : bool :=
  match e with
    | BTrue => true
    | BFalse => false
    | BEq a1 a2 => Zeq_bool (aeval a1) (aeval a2)
    | BLe a1 a2 => Zle_bool (aeval a1) (aeval a2)
    | BNot b1 => negb (beval b1)
    | BAnd b1 b2 => andb (beval b1) (beval b2)
  end.

```

et BFalse sont de type bool , ils correspondent à True et False qui quant à eux sont définit dans la sorte Prop .
Zeq_bool et Zle_bool correspondent à l'égalité *eq%Z et à l'infériorité Le%Z à ceci près qu'ils ont pour domaine* $\mathbb{B}$ *et non Prop . Le co-domaine reste inchangé :* $\mathbb{Z}^2$ *.*

```
Check(Zeq_bool).
// (* Zeq_bool: Z -> Z -> bool *)
```

Il en va de même pour negb et andb qui sont équivalent à neg et and excepté qu'ils ont pour co-domaine $\mathbb{B}^2$ *et pour domaine* $\mathbb{B}$*.*

2.5 Variable :

2.5.1 Système Déductif :

Un Système Déductif est constitué de :

- *Un alphabet*
- *Une grammaire*
- *Un schéma d'axiome (Ax)*
- *Un ensemble de règle d'inférence (Ri)*

Dans la suite de ce TP "les règles" d'un système désignera l'ensemble : $Ax \cup Ri$ *.*

Remarque :

Un système déductif est la forme "factorisé" d'une théorie T *, pour rappel une théorie est un ensemble de formule supposé close définit sur un langage L.*
Par factorisé j'entends , tout expression qui est dérivable par une suite de règle d'inférence à partir d'un axiome du système appartient à la théorie T *.*

2.5.2 Introduction :

Donner une certification de type se résume :

" Si je veux prouver que l'expression est de type Sequoia alors je dois construire un arbre Sequoia qui à pour racine mon expression ."

La croissance de l'arbre se fait de bas en haut en utilisant les règles du système déductif .

On appelera feuille l'ensemble des règles appartenant au schéma d'axiomes du système .Et on dira que ces feuilles cloturerent la croissance d'une branche , en ce sens que l'espace d'hypothèse étant vide il ne reste plus rien à prouver ce qui achève la "croissance" .

2.5.3 Le Système Séquoia :

Soit une règle Aurore appartenant aux règles d'inférences du système Séquoia définit comme suit :

```
Aurore:Sexp->Sexp
// (* avec Sexp l'abreviation de Sequoia expression *)
```

Sens de la règle : "Aurore s1" est une expression de type Sequoia ssi s1 est une expression de type Sequoia .

Admettons que le schéma d'axiome du système Sequoia n'est constituer que de l'axiome Pomme alors je dois courronnés chacune des branches de mon arbres de preuve avec des Pommes .

Soit l'axiome Pomme définit comme suit :

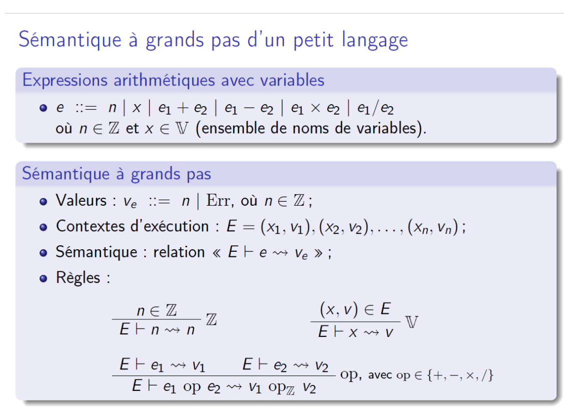
```
Pomme:Eexp->Sexp
//(* avec Eexp l'abreviation de Epicea expression *)
```

Sens de l'axiome : "Pomme c" est une expression de type Sequoia ssi c est une expression de type Epicéa.

D'après "Sens de l'axiome" , il faut égalemment que je prouve que l'expression que j'ai déclaré comme étant une Pomme est constitué par une unique expression de type Epicéa . Donc pour chaque feuille de mon Sequoia il faut que je construise un arbre/une preuve du type Epicéa ...

2.5.4 Exemple des expressions arithmétiques :

Les expressions arithmétiques sont similaire au système Séquoia , en ce sens qu'il faudrait théoriquement apporter une justification que $n \in \mathbb{Z}$ ou que $v \in \mathbb{V}$ lorsque l'on cloturera ses feuilles avec ANum et AVar resp :



Grâce au type Coq s'en chargera pour nous (? cf.inférence type/analyse statique).

2.5.5 Interprétation des variables et constantes

Quand nous écrivons des nombres ou des variables nous supposons qu'il y a une correspondance implicite entre nos symboles ,par exemple : $\{1, x\}$ et leur interprétation définit sur $\mathbb{Z}$: On peut exprimer cette correspondance avec les fonctions suivantes :

- $fi_{num} : Num \to \mathbb{Z}$
- $fi_{var} : Var \to \mathbb{Z}$

Remarque :

fi_{var} est communément appelé un état : il associe à chaque variable une valeur dans $\mathbb{Z}$.
fi_{num} est supposé déjà définit par coq, il interpréte le symbole/chaine de caractère str1 par sa valeur entière par exemple: $fi_{num}("1") = 1$ avec $1 \in \mathbb{Z}$.
FAUX : mais je le laisse

Ainsi si pour un symbole s fi_{var} ou fi_{num} est définit ce symbole est reconnu comme membre de $\mathbb{V}$ resp $\mathbb{Z}$. Nous avons donc trouvé la justification de type que l'on cherché .

2.5.6 Allocation des variables :

fi_{num} a été supposé avoir été crée par coq il faut maintenant implémenter fi_{var} :
Pour implémenter la correspondance entre nos variables et leur valeur nous utilisons une "stack" comme en programmation , schématiquement la stack est un ensemble de case mémoire dont chacune peut être dédié à une variable x .

Pour ce fait nous définissons le type inductif:

```
//(* une variable est un identifiant et un identifiant est un nombre nat  *)
Inductive id : Set :=
    Id : nat -> id.
```

Exemple :
Definition X1 := Id 7. Signifie que la case identifier par le numéro 7 est associé à X1

Pour récupérer la valeur associé à la case mémoire id nous définissions une fonction totale :

```
//state <=> fi_var
Definition state := id ->Z. //(* nous supposons devoir stocker que des entiers *)
```

L'initialisation de la stack se fait par une fonction empty_state qui associe la valeur 0 à toutes ses cases

```
//(* initialise la stack *)
Definition empty_state : state :=
  fun _ => 0.
```

La modification de la stack se fait par la fonction update .

```
//(* update la stack *)
Definition update (st : state) (X : id) (z : Z) : state :=
  fun (Xa : id) => if (beq_id X Xa) then z else st Xa.
```

Update empile des modifications de la stack ainsi si X1 fait l'objet de deux update seul le plus récent sera pris en compte (comportement FIFO) :

```
Definition X1 := Id 7.

Eval compute in (empty_state X1).  //(* X1:=0 *)

Eval compute in ((update empty_state X1 (-100)) X1). //(* X1:=100 *)

Eval compute in ((update (update empty_state X1 (-100)) X1 (-102)) X1). //(* overwrite X1
     *)

Definition X2 := Id 7.
```

```
10 Eval compute in ((update (update empty_state X1 100) X2 101) X1). //(* beq_id X1 X2 donc
     X1=X2 := 101 *)
11
12 Definition X3 := Id 8.
13 Eval compute in ((update (update empty_state X1 100) X3 101) X3). // (* X3:=101  *)
```

Prenons la ligne 5 :

1) On initialise la stack à 0 // (fun_0 (Xa : id) return 0 *)*
*2) La case 7 est rempli avec -100 (*fun_1 (Xa : id) return (Xa == X1 ? -100 : (fun_0 Xa)) *)*
*3) La case 7 est rempli avec -102 (*fun_2 (Xa : id) return (Xa == X1 ? -102 : (fun_1 Xa)) *)*
4) On demande : " qu'est ce qu'il y a dans la de X1 ?" ((fun_2 X1) *)*

5) Reponse : -102 .
Car fun_2(Xa : id) := return (Xa == X1 ? -102 : (Xa == X1 ?-100 :0))
Et donc substituer Xa par X1 dans fun_2(Xa : id) puis évaluer l'expression obtenue donne -102 .

2.5.7 Code Compléter :

```
//(* Syntaxe: *)

Inductive aexp : Type :=
  | ANum : Z -> aexp
  | AId : id -> aexp
  | APlus : aexp -> aexp -> aexp
  | AMinus : aexp -> aexp -> aexp
  | AMult : aexp -> aexp -> aexp


Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.

//(* Semantique : *)

Fixpoint aeval (st : state) (a : aexp) : Z :=
  match a with
    | ANum z =>  z
    | AId id => st id
    | APlus a1 a2 => (aeval st a1) + (aeval st a2)
    | AMinus a1 a2 => (aeval st a1) - (aeval st a2)
    | AMult a1 a2 => (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (e : bexp) : bool :=
  match e with
  | BTrue => true
  | BFalse => false
  | BEq a1 a2 => Zeq_bool (aeval st a1) (aeval st a2)
  | BLe a1 a2 => Zle_bool (aeval st a1) (aeval st a2)
  | BNot b1 => negb (beval st b1)
  | BAnd b1 b2 => andb (beval st b1) (beval st b2)
  end.
```

Finalement nous n'avons pas changé grand chose : renommé une règle pour mieux exprimer "l'implémentation" et rajouter l'état st .

2.6 Instructions :

```
Inductive instr_dlh : Set :=
  | ISkip : instr_dlh
  | IAss : id -> aexp -> instr_dlh
  | ISeq : instr_dlh -> instr_dlh -> instr_dlh
  | IIf : bexp -> instr_dlh -> instr_dlh -> instr_dlh
  | IWhile : bexp -> instr_dlh -> instr_dlh.
```

```
Inductive ceval : instr_dlh -> state -> state -> Prop :=
  | E_Skip : forall (st : state),
      (ceval SKIP  st  st)
  | E_Ass : forall (st : state) (a : aexp) (z : Z) (X : id),
       aeval st a = z ->  (ceval (X ::= a)  st  (update st X z))
  | E_Seq : forall (i1 i2 : instr_dlh) (st st' st'' : state),
       (ceval i1  st  st' ) -> (ceval i2  st' st'') -> (ceval (i1 ; i2)  st  st'')
  | E_IfTrue : forall (st st': state) (i1 i2 : instr_dlh ) (b : bexp),
       beval st b = true -> (ceval i1  st st') -> ( ceval (IFB b THEN i1 ELSE i2 FI)  st
    st')
  | E_IfFalse : forall (st st': state) (i1 i2 : instr_dlh ) (b : bexp),
       beval st b = false -> (ceval i2  st  st') -> (ceval (IFB b THEN i1 ELSE i2 FI) st
    st')
```

```
  | E_WhileEnd : forall (st : state) (i1 : instr_dlh ) (b : bexp),
      beval st b = false -> (ceval (WHILE b DO i1 END) st  st)
  | E_WhileLoop : forall (st st' st'': state) (i1 : instr_dlh) (b : bexp),
      beval st b = true ->
      (ceval i1  st st') ->
      (ceval (WHILE b DO i1 END)  st'  st'') ->
      (ceval (WHILE b DO i1 END) st  st'').
```

Voir Appendices pour la notation de M.Delahaye qui se veut plus esthétique .

3 Logique de Hoare :

3.1 Triplet d'Hoare

La logique de Hoare s'exprime sous la forme de triplet appelé triplet de Hoare ils sont de la forme : $\{P\}S\{Q\}$

Un triplet de Hoare est valide si et seulement si :
$\forall$ *st st' ,* $< S, st > \rightarrow st' \implies P\ st = tt \implies Q\ st' = tt$ *.*

Cela se note : $| = \{P\}S\{Q\}$ *Et* $< S, st > \rightarrow st'$ *veut dire : "Appliquer l'instruction S à l'état st résulte en l'état s' "*

Ainsi la Définition coq associé serait :

```
Definition Assertion := state -> Prop.

Definition hoare_triple (P : Assertion ) (S : instr_dlh) (Q : Assertion ) : Prop :=
  forall (st st':state),
    (ceval S  st  st')  ->
    P st ->
    Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q) (at level 90, c at next level).
```

Il faut comprendre le code suivant comme suit : " Si appliquer l'instruction S à l'état st résulte en l'état s' et que le prédicat P est vrai dans l'état st alors le prédicat Q est vrai dans le nouvel état st' "

Il en va de même pour la lecture du triplet $\{P\}S\{Q\}$ *signifie " si P est satisfait par un état st alors Q satisfait l'état st'" (avec st' l'état obtenue après avoir éxécuter l'instruction S dans st) .*

Remarque : tt et ff sont les valeurs de true resp false pour le type Prop ,elles peuvent aussi se noter False et True (cf. Curry Howard)

3.2 Règles d'inférences:

$$\frac{}{\{P\}\ \text{skip}\ \{P\}}\ \text{skip} \qquad \frac{}{\{P(e)\}\ x := e\ \{P(x)\}}\ :=$$

$$\frac{\{P\}\ i_1\ \{Q\} \qquad \{Q\}\ i_2\ \{R\}}{\{P\}\ i_1; i_2\ \{R\}}\ ;$$

$$\frac{\{P \wedge e\}\ i_1\ \{Q\} \qquad \{P \wedge \neg e\}\ i_2\ \{Q\}}{\{P\}\ \text{if}\ e\ \text{then}\ i_1\ \text{else}\ i_2\ \{Q\}}\ \text{if}$$

$$\frac{\{I \wedge e\}\ i\ \{I\}}{\{I\}\ \text{while}\ e\ \text{do}\ i\ \{I \wedge \neg e\}}\ \text{while}$$

$$\frac{\{P'\}\ i\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ i\ \{Q\}}\ \text{Aff}$$

Nous allons passé en revue chacune des règles ci dessus:

- *Axiome Skip : L'instruction Skip ne fait rien elle se note skip, ainsi le prédicat P reste valide après avoir éxécuter Skip sur l'état courant .*

- *Axiome Ass : L'instruction Ass assigne une valeur à une variable elle se note x:= a, ainsi il est correcte de dire que soit P un prédicat avec une variable libre x alors si P est vrai dans s après avoir substituer à x la valeur e ,noté :* $P[x < -e]*$ *alors P sera vrai après avoir affecté à x la valeur e dans s .*

  Remarque : Une formule/un prédicat peut être généraliser (cloture universelle) donc $P1 avecxlibre \equiv \forall x P1[x]$

  Un quantifieur universelle peut être remplacer par une fonction donc :

  $P1$ *avec x libre* $\leftrightarrow P(x : A) : Prop := (P1x)$ *(* ? Lam , pfouu oui pourquoi pas *)* *ainsi :*

$P1[x < -e] \leftrightarrow (Pe)$ *(* App *)*

La fonction P associe à tout x de type A (ici , Aexp) une représentation de P1 où x est substituer par la valeur passer à la fonction ,c'est bien l'exact concept d'une quantification universelle .

- *La règle Seq : L'instruction Seq exécute i1 puis i2 dans l'état courant elle se note i1;i2 ,"si on sait que i1 appliqué à l'état de départ donne s_i où Q est vrai et que i2 appliqué à s_i engendre l'état s' où R est vérifier " alors on a $\{P\}i1; i2\{\}$*

- *La règle If : L'instruction if éxécute i1 si e sinon elle éxecute i2 elle se note if e then i1 else i2 , je pense que c'est assez explicite .*

- *La règle While : L'instruction while éxecute i tant que e est évalué à vrai dans l'état courant elle se note while e do i , ce qui est décrit par le séquent : "si on sait que I et e sont vrai alors I reste vrai après avoir éxécuté i (quant à la valeur de e elle sera re-évaluer lors du prochain tour de boucle)"*

  I est appelé l'invariant car il est vrai après et avant l'éxécution de l'instruction , en d'autre termes il est vérifier avant et après chaque éxécution de i .

  e est la condition d'arrêt de la boucle , il est évalué à faux dans l'état final .

- *La règle Conseq: L'instruction Aff dit une chose très simple à savoir : si nous avons $\{P'\}i\{'\}$ et $P \implies P'$ et $Q' \implies Q$ alors on a $\{P\}i\{Q\}$ En d'autre termes nous pouvons affaiblir (d'où le diminutif Aff) la contrainte : $\{P'\}i\{Q'\}$, par affaiblir une contrainte j'entends ou renforcer la pre-condition ou affaiblir la post condition **

**: On dit affaiblir car le pouvoir déductif deviens moins fort :*
Soit $pre - condition \implies post - condition$ la contrainte :
Pour une pre-condition extrêmement contraignante , par exemple : pre condition = False cela n'impliquera rien sur la nature de la post condition car $False \implies post - condition$ est toujours vrai .
De même une post-condition trop faible : True n'imposera rien sur la pre-condition car $pre-condition \implies True$ est toujours vrai .
**:, si on est tatillons on devrait écrire $P[x < -A[e]]$ car e est une variable (lvalue) et donc elle se doit d'être intepréter pour devenir une valeur (rvalue) cf.voir syntaxe assignation .*

4 Preuve de Complétude et de Correction :

4.1 Introduction :

4.1.1 Logique premier ordre :

En logique du premier ordre :
- un système de règle est sémantiquement complet si toutes les formules satisfaitent par tous modèles M de T est prouvable/dérivable dans Sys . (semantic =¿ proof) :
$M| = TM| = F \implies Sys| - F$
- un système de règle est syntaxiquement complet si toutes formules dérivable à partir de Sys est satisfaite dans tous modèles M de T (proof =¿ semantic)
$Sys| - F \implies M| = TM| = F$
T est une théorie définit sur un langage (un ensemble de formule close) .
Sys est l'ensemble de règles syntaxiques associés à T .

4.1.2 Note Rappel

Note: J'appèle "modèle de T" un modèle qui satisfait toutes les formules de T ($\leftrightarrow$ satisfait toutes les règles de Sys , car Sys peut produire l'ensemble des mots de T) .
Rappel :

Un modèle est définit sur un alphabet A et permet d'évaluer la valeur sémantique d'une formule construite par une grammaire de A (=¿ la formule est bien formée) .Comme on l'a vue évaluer une expression se fait en interprétant l'ensemble des symboles : relation , fonction , constante , variables . Généralement on regroupe les intérprétation des ensembles R , F et C dans une structure et l'interprétation de l'ensemble V se fait via une "assignation". Ainsi on a la notation : M = ¡ I , assignation ¿

exemple : $A = , n, v_1, v_2, v_3$ *Soit M et F définit sur A tel que :* $I =< \mathbb{B}, \wedge, \neg >$ *assignation* $= \{v_1 \to 0, v_2 \to 0, v_3 \to 1\}$ *ainsi si* $F = v_3 n v_2$ *alors* $< I, assignation > | = F$

4.1.3 Logique Hoare:

Pour la logique de Hoare :

Nous avons déjà définit notre théorie : instr_dlh

Nous avons aussi notre structure d'intepértation en la présence de instr_dlh qui est un ensemble de "règles sémantiques" destinées a évaluer une formule . Notation:

$Sys_{sem} := \{E\_Skip, E\_Ass, E\_Seq, E\_ifTrue, E\_ifFalse, E\_WhileEnd, E\_WhileLoop\}$

Nous avons l'abstraction de multiple assignation grâce au type state (rappel :le state associe variable à valeur semantique) .

Nous avons égalemment un ensemble de règles syntaxique que l'on le notera Sys_{syn}

$Sys_{syn} := \{Skip, Ass, Seq, if, while, Aff\}$

Donc on peut dire

- le système de règle est sémantiquement complet si pour toutes les formules satisfaitent par tous les modèles satisfaisant toutes les règles de Sys_{sem} *la formule est dérivable par les règles de* Sys_{syn}

Soit A un alphabet et Mbf(A) l'ensemble des mots de A correctement formée ,

$\forall F \in Mbf(A) \forall st \in state \forall I \in struct, < I, st > | = Sys_{sem} \to < I, st > | = F \implies Sys_{syn} | - F$

- le système de règle est syntaxiquement complet si pour toutes les formules dérivables par les règles de Sys_{syn} *la formule est satisfaite par tous les modèles satisfaisant les règles de* Sys_{sem} *.*

$\forall F \in Mbf(A) \forall st \in state \forall I \in struct, Sys_{syn} | - F \implies < I, st > | = Sys_{sem} \to < I, st > | = F$

4.2 Notation:

Comme nous l'avons vue précédemment un triplet de Hoare est valide ssi :

$\forall$ *st st'* , $< S, st > \to st' \implies P\ st = tt \implies Q\ st' = tt$.

Pour la suite nous noterons cette formule :

$< I, st > | = \{P\}S\{Q\}$ *ou plus simplement* $| = \{P\}S\{Q\}$

Dans la seconde écriture le modèle :$< I, st >$ *est implicite tout comme l'état st et st' le sont pour la pre et post condition respectivement .*

Un triplet de Hoare est dérivable par les règles de Sys_{syn} *ce note :*

$Sys_{syn} | - \{P\}S\{Q\}$ *ou plus simplement* $| - \{P\}S\{Q\}$

4.3 Méthode :

Tout comme la preuve de correction et de complétude réaliser dans le cours 4-preuve de M.Delahaye nous raisonnerons par récurrence sur la taille de l'arbre de preuve .

Une formule à pour niveau 0 si l'arbre de preuve la précédent est de taille = 0 , typiquement les axiomes . Une formule à pour niveau n+1 si elle a été déduite par une règle du système déductif et que l'arbre de preuve de cette règle est de taille n .

4.4 Correction :

Dis grossièrement ,pour prouver la completude syntaxique nous devons vérifier que "toute expression dérivable à partir d'un axiome est vrai" , à savoir:

$| - \{P\}S\{Q\} \implies | = \{P\}S\{Q\}$

4.5 Les Axiomes :

La Base ou niveau 0 de la relation de récurrence se prouve en montrant que $|-\{P\}S\{Q\} \implies |=\{P\}S\{Q\}$ *est vrai pour les feuilles de notre arbre de preuve .*

- *Skip , nous avons* $\forall st, < skip, st > \to st$ *par la règle E_Skip prouvons que alors nous avons* $| = \{P\}skip\{P\}$ *en d'autre terme:*

  $\forall st, < skip, st > \to st \implies \forall$ *st st'* , $< S, st > \to st' \implies P\ st = tt \implies P\ st' = tt$.

  Alors supposons également que : $\forall$ *st st'* , $< S, st > \to st' \implies P\ st = tt$ *Et voyons si la formule P st' = tt est vrai , nous la noterons F .*

  Comme $\forall st, < skip, st > \to st$ *alors st=st' donc* $F \leftrightarrow F[st' < -st]$ *Ainsi F se réecrit P st = tt . Ce qui est vrai par hypothèse : Assumption .*

- *Ass , supposons que nous avons* $\forall st, < x := z, st > \to st[x \to (aevalstz)]$ *par la règle E_Ass prouvons que alors nous avons* $| = \{P[x \to (aevalstz)]\}Ass\{P\}$ *en d'autre terme:* $\forall$ *st* , $< x := z, st > \to st[x \to (aevalstz)] \implies (P[x \to (aevalstz)])\ st = tt \implies P\ st[x \to (aevalstz)] = tt$*.Supposons(P[x* $\to (aevalstz)])\ st = tt$ *et voyons si nous arrivons à* $Pst[x \to (aevalstz)] = tt$ *:* $(P[x \to (aevalstz)])$ *signifie que l'on subsitue les occurences de x dans P seront remplacé par la valeur sémantique de z dans st*

  $st[x \to (aevalstz)]$ *signifie que la valeur de x dans state sera la valeur sémantique de z .*

  Pst' *signifie alors que les occurence de x seront remplacé par sa valeur dans st'*

  donc si $st' = st[x \to (aevalstz)]$ *les occurences de x dans P seront remplacé par la valeur sémantique de z dans st*

  Donc : $(P[x \to (aevalstz)])st \leftrightarrow Pst[x \to (aevalstz)]$

  Ainsi comme par hypothèse : $(P[x \to (aevalstz)])st = tt$ *alors* $Pst[x \to (aevalstz)]$ *=tt . Fin .*

4.6 Les Règles d'inférences :

- *Seq , supposons que nous avons* $| = \{P\}i1\{Q\}$ *et* $| = \{Q\}i2\{R\}$ *prouvons alors que* $| = \{P\}i1; i2\{R\}$ *Or* $| = \{P\}i1; i2\{R\}$ *est :* $\forall$ *st* , $< i1; i2, st > \to st"$ *et P st = tt* $\implies$ *R s" = tt . Donc Supposons que nous avons :* $\forall$ *st st", P st =tt et* $< i1; i2, st > \to st"$ *et voyons si nous avons R st" = tt .*

  Si nous avons $< i1; i2, st > \to st"$ *alors nous avons un état st' tel que* $< i1, st > \to st'$ *and* $< i2, st' > \to st"$ *d'après la règle E_Seq.*

  Par $| = \{P\}i1\{Q\}$ *on a "* $< i1, st > \to st'$ *et P st = tt* $\implies$ *Q s' = tt " or on a aussi "*$< i1, st > \to st'$ *et P st = tt" donc on a Q s' =tt .*

  Par $| = \{Q\}i2\{R\}$ *on a "* $< i2, st' > \to st"$ *et Q st' = tt* $\implies$ *R s" = tt " or on a aussi "*$< i2, st' > \to st"$ *et Q st' = tt" donc on a R s" =tt .*

  Fin.

- *if , supposons que nous avons* $\forall$ *st* , $| = \{P \wedge (bevalste)\}i1\{Q\}$ *et* $| = \{P \wedge \neg(bevalste)\}i2\{Q\}$ *prouvons alors que* $| = \{P\}ifetheni1elsei2\{Q\}$

  Or $| = \{P\}ifetheni1elsei2\{Q\}$ *est :* $\forall$ *st* , $< ifetheni1elsei2, st > \to st'$ *et P st = tt* $\implies$ *Q s' = tt .*

  Donc Supposons que nous avons : $\forall$ *st* , $< ifetheni1elsei2, st > \to st'$ *et P st = tt et voyons si nous avons Q st' = tt .*

  Si nous avons $< ifetheni1elsei2, st > \to st'$ *alors nous avons soit* $< i1, st > \to st'$ *si* $(bevalste) = tt$ *soit* $< i2, st > \to st'$ *si* $(bevalste = ff$ *d'après la règle E_ifTop et E_ifBot .*

  Si nous sommes dans le premier cas : $(bevalste) = tt$ *Par* $| = \{P \wedge (bevalste)\}i1\{Q\}$ *on a "* $< i1, st > \to st'$ *et* $P \wedge (bevalste)$ *st = tt* $\implies$ *Q st' = tt " or on a aussi* $< i1, st > \to st'$, $(bevalste) = tt$ *et P s =tt par hypothèse du if donc on a Q st'=tt*

  Si nous sommes dans l'autre cas : $(bevalste) = ff$ *Par* $| = \{P \wedge \neg(bevalste)\}i2\{Q\}$ *on a "* $< i2, st > \to st'$ *et* $P \wedge (bevalste)$ *st = ff* $\implies$ *Q st' = tt " or on a aussi* $< i2, st > \to st'$, $(bevalste) = ff$ *et P s =tt par hypothèse du if donc on a Q st'=tt*

- *while , supposons que nous avons* $\forall$ *st ,* $| = \{I \wedge (bevalste)\} i \{I\}$ *prouvons alors que* $| = \{I\} whileedoi \{I \wedge \neg(bevalste)\}$

  Or $| = \{I\} whileedoi \{I \wedge \neg(bevalste)\}$ *est :* $\forall$ *st ,* $< whileedoi, st > \rightarrow$ *st" et I st = tt* $\implies$ $I \wedge \neg(bevalst"e)$ *st" = tt .*

  Donc Supposons que nous avons $\forall$ *st ,* $< whileedoi, st > \rightarrow$ *st" et I st = tt*

  Si nous avons $< whileedoi, st > \rightarrow$ *st" alors nous avons soit "* $< i, st > \rightarrow st'$ *et* $< whileedoi, st' > \rightarrow$ *st" si* $(bevalste)$ *s = tt soit* $< whileedoi, st > \rightarrow st'$ *si* $(bevalste)$ *s = ff .*

  Si nous sommes dans le premier cas : $(bevalste) = tt$ *alors on a :* $< i, st > \rightarrow st'$ *et* $< whileedoi, st' > \rightarrow$ *st" Par* $\forall$ *st ,* $| = \{I \wedge (bevalste)\} i \{I\}$ *on a "* $\forall$ *st ,* $< i, st > \rightarrow st'$ *et* $(I \wedge (bevalste))$ *st = tt* $\implies$ *I st' = tt " or on a* $< i, st > \rightarrow st'$ *par hypothèse du cas mais aussi I st =tt par hypothèse Donc on a I st' =tt .Ainsi on a* $< whileedoi, st' > - >$ *st" et I st' = tt Or par induction on si* $< whileedoi, st' > - >$ *st" et I st' = tt alors* $I \wedge \neg(bevalst"e)$ *st" =tt Donc on a* $I \wedge \neg(bevalst"e)$ *st" =tt Fin .*

- *cons, supposons que nous avons* $\forall$ *st ,* $| = \{P'\} i \{Q'\}$ *,* $P \implies P'$ *et* $Q' \implies Q$ *prouvons alors que* $| = \{P\} i \{Q\}$

  Or $| = \{P\} i \{Q\}$ *est :* $\forall$ *st ,* $< i, st > \rightarrow st'$ *et P st = tt* $\implies$ *Q st' = tt .*

  Donc Supposons que nous avons : $\forall$ *st ,* $< i, st > \rightarrow st'$ *, P st = tt et voyons si nous avons Q st' = tt .*

  Comme nous avons P s = tt or par supposition $P \implies P'$ *donc P' s =tt , nous avons aussi supposé que* $| = \{P'\} i \{Q'\}$ *est : "*$\forall$ *st ,* $< i, st > \rightarrow st'$ *et P' st = tt* $\implies$ *Q' st' = tt " donc comme on a* $< i, st > \rightarrow st'$ *et P' s =tt alors on a Q' st' =tt or par supposition* $Q' \implies Q$ *donc Q st' = tt .*

  Fin.

4.7 Complétude :

Dis grossièrement ,pour prouver la completude sémantique nous devons vérifier que "toute expression vrai est dérivable d'un axiome " , à savoir:

$| = \{P\} S \{Q\} \implies | - \{P\} S \{Q\}$

Par induction nous avons $| = \{P\} i1 \{Q\} \implies | - \{P\} i1 \{Q\}$

$\forall$ *st st',* $< i1, st > \rightarrow st'$*et st"* $< i2, st' > \rightarrow$ *st"*

**:(si elle n'est pas close il suffirait d'associer une assignation des variables libre à la structure où de la généraliser : cloture universelle)*

Coq se base sur le calcul des constructions , chaque ensemble est construit suivant un ensemble de briques atomiques (sa base) et d'expression complexes. Ainsi pour vérifier l'appartenance d'une expression à un ensemble il faut procéder comme suit:

```
// E.b_cst : est la base de E
//E.f_cst : sont les fonctions de E
Relation_appartenance ( expr:Type) : Bool {
if( appartient(expr ,E.b_cst) ) return true ;
else if (appartient(expr , E.f_cst )) return Relation_appartenance(expr')
else return false;
}
```

Pour pouvoir évaluer les expressions nous avons besoins de définir leur sémantique : En ce qui concerne les expressions booléenne

<u>Notation</u>: pour la suite de ce TP un ensemble d'instructions sera appelé un programme et le type instructions sera noté Stm (en référence à "statement") .

```
#include <stdio.h>
int main(void)
{
    printf("Hello World!");
}
```

References

[1] Coq Art (V8) , *Yves.Bertot, 2015,* https://www.labri.fr/perso/casteran/CoqArt/

[2] Software Foundations :Logical Foundation , *Benjamin C. Pierce,* https://softwarefoundations.cis.upenn.edu/lf-current/toc.html

[3] Certified Programming with Dependent Type , *Adam Chlipala ,* http://adam.chlipala.net/cpdt/

[4] Semantics with applications , A formal introduction (course) , *Nielson et Nielson, 2019.,* https://www.cs.ru.nl/~herman/onderwijs/semantics2019/

[5] Lambda Calculus , *wikipedia ,2020 ,* https://en.wikipedia.org/wiki/Lambda_calculus

[6] Coq Doc ,*Pierre Courtieu,* http://cedric.cnam.fr/~courtiep/downloads/loglangcoq/toc.html

[7] A Tutorial on Reflecting in Coq the generation of Hoare proof obligations ,*Sylvain Boulmé,* http://www-verimag.imag.fr/~boulme/HOARE_LOGIC_TUTORIAL/index.html

[8] Gentle Introduction to Dependent Type , *Boro Sitnikovski,2018,* https://www.researchgate.net/publication/341780951_Gentle_Introduction_to_Dependent_Types_with_Idris

[9] Cours MPRI Coq , *Christine Paulin,* https://www.lri.fr/~paulin/DEA/introduction.html https://www.lri.fr/~paulin/MPRI/notes/

[10] Software Foundations :Logical Foundation (solution) ,*Junyoung Clare Jang, Ailrun (maintainer),* https://github.com/Ailrun/software_foundations_solution

[11] Introduction à coq , *Micaela Mayero ,Paris 13 ,* https://lipn.univ-paris13.fr/~mayero/?content=master

[12] Logique Hoare (AGREG) ,*Pierre Le Barbenchon ,Rennes ,* http://perso.eleves.ens-rennes.fr/people/pierre.le-barbenchon/agreg.html

[13] Cours M2 coq , *Pierre Courtieu,* http://cedric.cnam.fr/~courtiep/

[14] Outils de preuve et verification , *Pierre Courtieu, 2008.,*

[15] A Tutorial on Reflecting in Coq the generation of Hoare proof obligations (github) , *Kartik Singhal (maintainer) ,* https://github.com/coq-community/hoare-tut

[16] Software Foundations :Logical Foundation (solution 2) ,*Haklabbeograd (maintainer),* https://github.com/haklabbeograd/software-foundations-coq-workshop

[17] Video Coq , *Yves.Bertot, 2017.,* http://www-sop.inria.fr/members/Yves.Bertot/videos-coq/

[18] Logique equationnelle et reecriture , *Sophie Pinchinat (et David Cachera), 2017.,* https://people.irisa.fr/Sophie.Pinchinat/

[19] Pedagogical prover of ENS Rennes , *IRISA,* http://pravda.irisa.fr/

[20] Complément Recherche : Logique equationnelle et evaluation symbolique , *Marc Aiguier,* https://perso.ecp.fr/~aiguierm/publications/support-cours/

[21] Lambda-calculus Types and Models , *Jean-Louis Krivine ,(trad. René Cori) ,* https://www.irif.fr/~krivine/articles/

Appendices

```
// (* Nous definissons beq_id qui comme sont nom l'indique renvoie un booleen et
    correpond a la relation eq pour le type id *)

Definition beq_id (X1 : id) (X2 : id) : bool :=
    match (X1, X2) with
      (Id x1, Id x2) => beq_nat x1 x2 //(* si X1 et X2 sont le resultat de la fonction Id
     applique a x1 et x2 resp alors beq_id X1 X2 <-> beq_nat x1x2 *)
    end.
```

```
Notation "'SKIP'" := ISkip.
Notation "X '::=' a" :=  (IAss X a) (at level 60). //(* ':=' est deja pris *)
Notation "i1 ; i2" := (ISeq i1 i2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c " := (IWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 " :=(IIf e1 e2 e3) (at level 80, right
    associativity).//(* 'IF' est deja pris *)


Reserved Notation "E '|-' i1 '~~>' E'" (at level 40 ).  //(* le reserved pour le where *)

Inductive ceval : instr_dlh -> state -> state -> Prop :=
  | E_Skip : forall (E : state),
      E |- ISkip   ~~> E
  | E_Ass : forall (E : state) (e : aexp) (v : Z) (x : id),
        (aeval E e = v) ->  E |- x ::=  e  ~~>  (update E x v)
  | E_Seq : forall (i1 i2 : instr_dlh) (E E1 E2 : state),
        ( E |- i1  ~~>E1 ) -> (E1 |- i2  ~~> E2) -> E |- i1 ; i2 ~~>  E2
  | E_IfTrue : forall (E E': state) (i1 i2 : instr_dlh ) (e : bexp),
        (beval E e = true) -> (E |- i1 ~~> E') ->  E |- IFB e  THEN i1 ELSE i2 ~~> E'
  | E_IfFalse : forall (E E': state) (i1 i2 : instr_dlh ) (e : bexp),
        (beval E e = false) -> (E |- i2  ~~>  E') -> E |- IFB e THEN  i1 ELSE i2  ~~> E'
  | E_WhileTop : forall (E : state) (i : instr_dlh ) (e : bexp),
      (beval E e = false ) -> E |- WHILE e  DO i  ~~>  E
  | E_WhileBot : forall (E E' E'': state) (i : instr_dlh) (e : bexp),
        beval E e = true ->
        (E |- i  ~~> E') ->
        (E' |- WHILE e  DO i  ~~>  E'') ->
        E |- WHILE e DO i  ~~>  E''
  where "E '|-' i '~~>' E'" := (ceval i E E').
```