Master Parisien de Recherche en Informatique 2008–2009
Assistants de Preuve

**TD/TP 8 - Programmation impérative**

Mardi 27 janvier 2009

# 1 Utilisation de Why

Les binaires pour Why et Alt-Ergo sont disponibles dans `~paulin/bin` que vous pouvez temporairement ajouter à votre variable `path`.

Pour utiliser Why, on peut créer un fichier `f.why` puis utiliser l'interface graphique `gwhy f.why` qui permet de visualiser les obligations de preuve et de les prouver automatiquement.

Pour utiliser Coq il faut d'abord faire `why -coq f.why` qui engendrera le fichier Coq `f_why.v` contenant les obligations de preuve.

# 2 McCarthy's 91 function

McCarthy's 91 function is the function $f$ from $\mathbb{Z}$ to $\mathbb{Z}$ defined by

$$\begin{cases} f(n) = f(f(n+11)) & \text{if } n \leq 100 \\ f(n) = n - 10 & \text{otherwise.} \end{cases}$$

1. Define function $f$ in Why. The Why syntax for a recursive function is

   ```
   let rec f (n:int) : int = ...
   ```

2. Annotate $f$ in order to prove that $f(n)$ is 91 when $n \leq 100$ and $n - 10$ otherwise.
3. Prove the termination of $f$ by inserting the following variant

   ```
   let rec f (n:int) : int { variant max(0,101-n) } = ...
   ```

   Since `max` is not a primitive function, you must introduce it with a **logic** and axiomatize it with an **axiom**.

# 3 Fibonacci Function

1. Introduce the Fibonacci function $F$ with a **logic** and three **axiom**s. We recall that $F(0) = F(1) = 1$ and $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$.
2. Define a recursive function $f_1$ computing $F$ (with a naive, *i.e.* exponential, algorithm). Prove its correctness and termination.
3. Define a function $f_2$ computing $F$ using a linear algorithm which maintains $F(n-1)$ and $F(n)$ in two references. Prove its correctness and termination.
4. Define a third function $f_3$ computing $F(n)$, using the same linear algorithm but using a recursive function instead of a loop. Note how the loop invariant is naturally transformed into a precondition.

# 4 Preuve de programme (Examen 2003-2004)

Dans cet exercice, on s'intéresse à la preuve de programmes contenant des boucles for « à la Caml », c.-à-d. de la forme `for` $i = e_1$ `to` $e_2$ `do` $e_3$ `done` avec la sémantique suivante :

- $e_1$ et $e_2$ sont évalués en $v_1$ et $v_2$ une fois pour toutes ;
- si $v_1 > v_2$ on ne fait rien ;
- sinon, on évalue $e_3$ successivement pour $i$ prenant les valeurs $v_1$, $v_1 + 1$, ..., $v_2$.

Remarquons que $i$ n'est visible que dans $e_3$, dans lequel il n'est pas modifiable.

On rappelle qu'en Coq, `Z` est le type des entiers relatifs, et on dispose d'une fonction `Z_of_nat` d'injection de `nat` dans `Z`. On suppose également donnée une fonction inverse `nat_of_Z`, qui envoie tous les négatifs sur 0.

1. Définir une fonction Coq, de type `Z->Z->Z`, équivalente au programme Caml suivant :

   ```
   let f a b =
     let d = ref 1 in
     for i=a to b do d := 19 * !d + i done;
     !d
   ```

   On utilisera obligatoirement une fonction auxiliaire définie par récurrence structurelle sur un `nat`.

2. Compléter les règles de logique de Hoare suivantes, en justifiant brièvement pourquoi celles-ci sont correctes.

   $$\frac{}{\{(a > \ldots) \wedge \ldots\}\ \texttt{for}\ i = a\ \texttt{to}\ b\ \texttt{do}\ s\ \texttt{done}\ \{Q\}}$$

   $$\frac{\{(\ldots \leq i \leq \ldots) \wedge I(i)\}\ s\ \{I(\ldots)\}}{\{a \leq \ldots \wedge I(\ldots)\}\ \texttt{for}\ i = a\ \texttt{to}\ b\ \texttt{do}\ s\ \texttt{done}\ \{I(\ldots)\}}$$

   Attention : dans ces règles, $a$ et $b$ sont des constantes entières, et non des expressions.

3. Pour prouver en Coq des programmes avec boucles for, on a besoin d'un principe de récurrence `for_rec` sur un intervalle $[a, b]$ d'entiers, de type

   ```
   forall (a b:Z), a <= b+1 ->
     forall (P : Z -> Set),
       P a -> (forall i, a <= i <= b -> P i -> P (i+1))
       -> P (b+1).
   ```

   On demande de construire une définition de `for_rec`. On pourra utiliser une fonction auxiliaire travaillant sur des `nat`, de manière analogue à la première question (et dans ce cas on aura soin de bien préciser le type de cette fonction). On s'autorisera à ne pas donner de preuves pour les propriétés purement logiques (i.e. de sorte `Prop`), on utilisera dans de tels cas la notation $(? : P)$ où $P$ est le type attendu.

4. À l'aide de `for_rec`, donner une définition Coq d'une fonction `sqr` de type

   ```
   forall z:Z, z>=0 -> { s:Z | s=z*z }
   ```

   correspondant au programme Caml

   ```
   let sqr z =
     let s = ref 0 in
     for i=0 to z-1 do s := !s + 2*i + 1 done;
     !s
   ```