Projet Why3

Mickaël LAURENT

1 Implémentation et preuves

0. Axiomes Je n'ai eu besoin que de l'axiome proposé dans le sujet :

$$\forall xy, 0 < x < y \implies log(x) < log(y)$$

1. power2 J'avais initialement implémenté cette fonction de manière itérative (code toujours présent mais commenté), mais j'ai finalement opté pour une version récursive, à la fois plus concise à implémenter et à prouver. En effet, la bibliothèque standard définie elle-même power de manière récursive, rendant la preuve de mon implémentation immédiate. Il a seulement été nécessaire d'indiquer le variant : l'argument l en entrée de la fonction.

2. shift_left J'ai opté pour l'implémentation suivante : `z * (power2 l)`.
Elle traduit littéralement la spécification de la fonction : `result = z * (power 2 l)`.

3. ediv_mod Mon implémentation est itérative et utilise la méthode d'euclide, comme celle faite en exercice durant l'année. En revanche, ici x peut être négatif, ainsi j'utilise deux implémentations différentes selon le cas : si $x \geq 0$ alors je commence avec un reste r positif auquel je soustrait y à chaque itération, sinon, si $x < 0$, je commence avec un r négatif auquel j'ajoute y à chaque itération.
La terminaison se prouve facilement en spécifiant le variant : r dans un cas, $-r$ dans l'autre.
La correction se prouve également sans difficulté en utilisant l'invariant adéquate, par exemple pour le cas $x < 0$: `x = !q * y + !r /\ !r < y`.

4. shift_right J'utilise l'implémentation suivante : `let d,_ = ediv_mod z (power2 l) in d`, qui traduit directement la spécification voulue : `result = ED.div z (power 2 l)`.

5. isqrt Là encore j'utilise l'implémentation itérative vue en cours :

```
let count = ref 0 in
let sum = ref 1 in
while !sum <= n do
        invariant { 0 <= !count /\ !count * !count <= n /\ !sum = (!count+1)*(!count+1) }
        variant { n - !sum }
        count := !count + 1;
        sum := !sum + 2 * !count + 1
done ; !count
```

La terminaison est immédiate étant donné le variant, mais la correction (`result = floor (sqrt (from_int n))`) nécessite tout de même de spécifier quelques trivialités après la boucle while (voir dans le code).

NOTE : je n'ai pas eu besoin du lemme suggéré (euclid_uniq) pour les 5 questions précédentes.

6-12. some properties La quasi-totalité des propriétés de l'énoncé ont été prouvées immédiatement par Z3, excepté `_B_9` et `_B_11` qui ont nécessité une étape intermédiaire (lemmes `_B_9_aux` et `_B_11_aux`).

```
lemma _B_6 : forall n. _B n >. 0.
lemma _B_7 : forall n, m. (_B n) *. (_B m) = _B (n + m)
lemma _B_8 : forall n. _B n *. _B (-n) = 1.
lemma _B_9_aux : forall n. sqrt (_B (2*n)) = _B n
lemma _B_9 : forall a,n. 0. <=. a -> sqrt (a *. _B (2*n)) = (sqrt a) *. (_B n)
lemma _B_10 : forall y. 0 <= y -> _B y = from_int (power 4 y)
lemma _B_11_aux : forall y. (pow b y)*.(pow b (-.y)) = 1. (* Same as B_8 but for reals *)
lemma _B_11 : forall y. y < 0 -> _B y = inv (from_int (power 4 (-y)))
lemma _B_12 : forall y. 0 <= y -> power 2 (2 * y) = power 4 y
```

13. framing La définition de l'énoncé est meilleure que $|x - p(B^{-n})| < B^{-n}$ car elle est divisée en deux inégalités plus simples que l'on peut prouver séparemment. Ces deux inégalités auraient de toute façon dû être prouvées avant de pouvoir prouver $|x - p(B^{-n})| < B^{-n}$ car une disjonction de cas est requise par la valeur absolue.

2 Conclusion

1. Lemmes en dehors des fonctions : plus efficace pour prouver des résultats mathématiques indépendants.