Les nombres premiers au crible de la preuve formelle

Josué Moreau

▶ **To cite this version:**

Josué Moreau. Les nombres premiers au crible de la preuve formelle. JFLA 2021 - 32 èmes Journées Francophones des Langages Applicatifs, Apr 2021, Virtual, France. pp.210-219. hal-04757088

HAL Id: hal-04757088
https://hal.science/hal-04757088v1

Submitted on 28 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Les nombres premiers au crible de la preuve formelle

Josué Moreau*

Université Paris-Saclay

Résumé

Dans cet article, nous nous intéressons à la vérification formelle du crible d'Euler, un algorithme permettant d'énumérer tous les nombres premiers inférieurs à une certaine borne. Après avoir décrit cet algorithme ainsi qu'une implémentation efficace de celui-ci en OCaml, nous présentons la preuve de celle-ci, réalisée à l'aide de Why3. Nous examinons également l'efficacité du code prouvé par rapport à celle d'autres cribles, tel que celui d'Ératosthène.

1 Introduction

Le crible d'Euler, tout comme le crible d'Ératosthène, permet d'énumérer l'ensemble des nombres premiers compris entre 2 et une limite donnée N. Le crible d'Euler a une complexité temporelle en $O(N)$ pour énumérer tous les nombres premiers dans l'ensemble $\{2, ..., N\}$ [8] alors que celui d'Ératosthène a une complexité en $O(N \log \log N)$.

Le crible d'Euler est plus difficile à implémenter que le crible d'Ératosthène. C'est d'autant plus vrai si on réalise quelques optimisations pour rendre le crible d'Euler plus efficace en temps et en mémoire. Ces subtilités d'implémentation sont autant d'occasions de faire des erreurs, d'où la nécessité de vérifier formellement l'algorithme.

Dans cet article, on décrit une telle vérification réalisée à l'aide de l'outil Why3 [2]. La preuve a été réalisée uniquement à l'aide de démonstrateurs automatiques. Cependant, une partie de cette preuve a nécessité l'utilisation de différents outils fournis par Why3 permettant d'aider le raisonnement des démonstrateurs automatiques. L'intégralité de cette preuve est disponible en ligne [1].

Cet article est organisé de la façon suivante. La section 2 décrit l'algorithme du crible d'Euler, ainsi que son implémentation dans le langage OCaml. Puis la section 3 décrit les grandes lignes de la preuve qui a été réalisée à l'aide de Why3. Enfin, la section 4 présente le code OCaml automatiquement extrait par Why3 à partir de la preuve qui a été réalisée. En particulier, nous comparons l'efficacité de ce programme extrait avec des implémentations du crible d'Ératosthène et du crible d'Ératosthène segmenté.

2 Le crible d'Euler

Le code de l'implémentation, dans le langage OCaml, de l'algorithme que nous allons prouver se trouve dans la figure 2. L'algorithme du crible d'Euler prend comme unique entrée la limite $N \geq 2$ et va renvoyer tous les nombres premiers compris entre 2 et N. Il se divise en deux parties. La première partie prend un nombre n non marqué tel que $2 \leq n \leq N$ et va marquer tous les multiples np tels que $2 \leq np \leq N$ et $p \geq n$ n'est pas marqué. Elle correspond à la fonction `remove_products` du code OCaml. La deuxième partie de l'algorithme boucle sur un entier n et appelle à chaque tour de boucle la fonction `remove_products` puis affecte n au plus

*Ce travail a été effectué dans le cadre d'un stage de L3 à l'Université Paris-Saclay, du 1er au 30 juin 2020, encadré à distance par Jean-Christophe Filliâtre.

petit entier inférieur à la limite, non marqué et strictement supérieur à n et recommence tant que $n \leq N$.

À la fin de chaque étape de la boucle principale, tous les produits de n sont marqués. En effet, en supposant que pour tout $2 \leq k < n$, les multiples de k sont déjà marqués, l'appel de `remove_products` marque tous les produits np tels que $p \geq n$ n'est pas marqué. Si un $p \geq n$ était déjà marqué, alors, par définition du marquage, il existe un i non marqué et j tel que $2 \leq i < n$, $2 \leq j < p$ et $ij = p$. Or, comme tous les multiples de i étaient déjà marqués avant l'appel de `remove_products`, $ijn = pn$ a déjà été marqué, au plus tard lors de l'appel de `remove_products` sur l'entier i. Ainsi, l'algorithme ne marquant que les produits de nombres non marqués, il ne marque qu'une seule fois chaque entier non premier, comme multiple de son plus petit facteur premier. C'est là la clé de la complexité linéaire du crible d'Euler.

L'implémentation de l'algorithme que nous allons prouver utilise une liste chaînée croissante pour représenter les nombres de l'ensemble $\{2, ..., N\}$. Cette liste est simplement chaînée et est implémentée dans un tableau des suivants `arr`. Une liste chaînée est importante pour l'efficacité car elle permet d'obtenir le prochain entier à parcourir en temps constant.

Voici un schéma décrivant l'état de la liste chaînée dans le tableau `arr` à un moment donné pendant une exécution du crible d'Euler. Dans celle-ci l'algorithme vient de marquer (en gris) tous les multiples de 2 et de 3. Il va commencer à marquer les multiples de 5, en marquant les produits de 5 avec 5, 7, 11, 13, ...

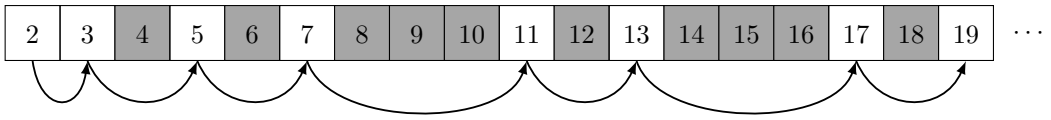

FIGURE 1 – Illustration de la liste chaînée.

Lors du marquage d'un entier i, on ne le supprime pas de la liste chaînée car cela coûterait trop cher en temps, ou nécessiterait une liste doublement chaînée qui coûterait trop cher en espace. Au lieu de cela, on se contente de changer le signe de `arr[`i`]` pour signifier le marquage de i. Lorsque la fonction `remove_products` parcourt la liste pour barrer tous les multiples de n, elle en profite pour éliminer les entiers qui sont marqués. Pour cela, elle maintient l'élément précédent dans la liste dans une variable `p`. Il est donc important de remarquer qu'un entier i, au moment où la boucle de `remove_products` arrive sur lui, peut très bien être déjà marqué. Si c'est le cas, alors il aura nécessairement été marqué par une précédente itération de la boucle. Lorsque la boucle de `remove_products` arrive sur l'entier $i =$ `arr[p]`, elle marque ni. Puis, si i était marqué, elle modifie `arr[p]` en lui donnant pour valeur `arr[`i`]` puis elle recommence en regardant l'entier `arr[p]`. Si i n'était pas marqué, elle recommence en regardant l'entier `arr[`i`]`. Cette opération permet donc à la liste chaînée de "sauter" par dessus les entier déjà marqués, et donc d'examiner au plus une fois chaque entier marqué.

Avec ce procédé, il y a un nombre important d'entiers qui sont marqués et qui ne seront pas réexaminés par la fonction `remove_products` et qui, par conséquent, seront encore dans la liste chaînée à la fin de l'algorithme. Il s'agit des entiers qui ont été marqués comme étant multiples d'un certain n, mais dont le produit avec tous les entiers suivant n dans la liste chaînée est supérieur à la limite du crible. Ces entiers seront supprimés lors de l'extraction des nombres premiers de la liste chaînée.

```
(* on représente une liste chaînée par un tableau, chaque indice du tableau *)
(* contient l'indice de la case suivante dans la liste chaînée. *)
type t = { arr: int array;  max: int;  max_arr: int; }

(* marque tous les multiples de n encore non marqués et saute par dessus les *)
(* nombres déjà marqués qu'il croise *)
let remove_products (t: t) (n: int) : unit =
  let d = get_max t / n in
  let rec loop (p: int) : unit =
    let next = get_next t p in
    if 0 <= next && next <= get_max t then
      if next <= d then begin
        set_mark t (n * next);
        if get_mark t next then begin
          (* si next était déjà marqué, alors on le retire de la liste chaînée *)
          (* en sautant par dessus *)
          set_next t p (get_next t next);
          loop p
        end else loop next (* sinon on passe à l'entier suivant *)
      end in
  set_mark t (n * n); (* tous les n * i pour i < n sont déjà marqués *)
  loop n

let euler_sieve (max: int) : int array =
  (* initialement, la liste chaînée contient tous les nombres impairs et pour *)
  (* marquer la fin de la liste chaînée, son dernier entier a pour valeur max + 1 *)
  let t = create max in
  let rec loop (n: int) : unit =
    (* n prend successivement les valeurs des nombres premiers *)
    remove_products t n;
    let nn = get_next t n in
    if nn <= max / nn then loop1 nn in
  if max >= 9 then loop 3;
  ... extraction des nombres premiers ...
```

FIGURE 2 – Code du crible d'Euler en OCaml.

Afin de diviser l'espace utilisé par deux, le tableau `arr` ne représente que les nombres impairs. Ainsi, la case d'index i du tableau représente l'entier $2i+1$. Les fonctions `get_next` et `set_next` utilisées dans le code précédent sont donc les suivantes :

```
let set_next (t: t) (i: int) (v: int) : unit = t.arr.(i / 2) <- v
let get_next (t: t) (i: int) : int =
  if t.arr.(i / 2) < 0 then - t.arr.(i / 2)
  else t.arr.(i / 2)
```

De plus, pour chaque indice i, `arr[`i`]` contient un entier positif si et seulement si l'entier i est non marqué. Dans ce cas, `arr[`i`]` est l'entier suivant i dans la liste chaînée. L'opération de marquage d'un entier i consiste à remplacer le contenu de la case i par son opposé. On a donc dans le code les fonctions `get_mark` et `set_mark` suivantes :

```
let set_mark (t: t) (i: int) : unit =
  if t.arr.(i / 2) >= 0 then t.arr.(i / 2) <- - t.arr.(i / 2)
let get_mark (t: t) (i: int) : bool = t.arr.(i / 2) < 0
```

Enfin, la fonction `create` ci-dessous crée l'état initial de la structure de donnée maintenue par l'algorithme. La liste chaînée décrite précédemment contient uniquement les entiers impairs. Il y a donc dans chaque case i du tableau `arr` l'entier $2i + 3$, à l'exception de la dernière case du tableau, qui représente le dernier entier de la liste chaînée, qui contient l'entier `max` $+ 1$.

```
let create (max: int) : t =
  let len_arr = (max - 1) / 2 + 1 in
  let arr = Array.make len_arr (-2) in
  for i = 1 to len_arr - 1 do
    arr.(i) <- if i = len_arr - 1 then max + 1 else 2 * i + 3
  done;
  { arr = arr; max = max; max_arr = (max - 1) / 2 }
```

Voici une autre figure décrivant l'implémentation du tableau d'entiers, représentant uniquement les entiers impairs (notés entre parenthèses, en dessous de leurs indices respectifs dans le tableau) et le marquage avec les entiers négatifs. Les flèches sont dessinées ici uniquement pour montrer les liaisons réalisées par ce tableau. Les multiples de 3 ont déjà été marqués et ceux qui se trouvent dans la portion du tableau représentée ont été éliminés de la liste chaînée. Les multiples de 5 sont en cours de marquage. On peut en effet constater que 25 a été marqué mais pas encore éliminé de la liste chaînée. Il le sera lorsque l'algorithme marquera 5×25.

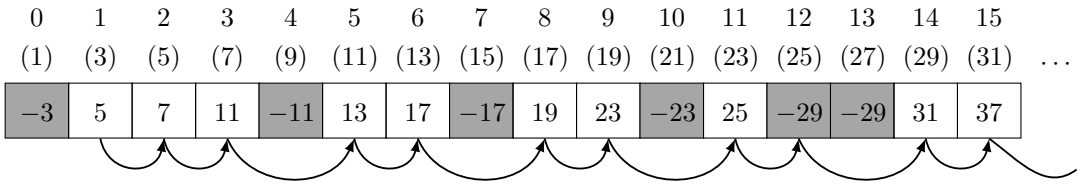

FIGURE 3 – Illustration de l'implémentation de la liste chaînée.

Enfin, à la toute fin de l'algorithme, on souhaite renvoyer un tableau ne contenant que les nombres premiers. Pour cela, on pourrait commencer par compter les nombres premiers de la liste chaînée, puis créer un tableau de cette taille et enfin y copier les nombres premiers. Cependant, on peut faire un peu mieux, en écrasant le début du tableau contenant la liste chaînée avec les nombres premiers, puis en extrayant avec `Array.sub` le préfixe contenant tous les nombres premiers. Le code de la figure 4, qui correspond à la fin du code de la figure 2, représente cette extraction des nombres premiers.

Dans l'implémentation OCaml que nous allons prouver, on manipule à plusieurs reprises un type `t`. Ce type contient comme premier champ le tableau `arr` décrit précédemment. Le champ `max` correspond à la limite donnée en entrée au crible et le champ `max_arr` correspond au plus grand indice du tableau `arr`, c'est-à-dire `max_arr` $\stackrel{\text{def}}{=} \frac{\texttt{max}-1}{2}$. La fonction `remove_products` du code OCaml correspond à la fonction du même nom décrite précédemment. La fonction `euler_sieve` est la fonction principale du crible. Elle appelle `remove_products` dans sa boucle principale et extrait ensuite les nombres premiers de la liste chaînée pour les renvoyer dans un tableau.

```
let cnt = ref 1 in
let p = ref 1 in t.arr.(0) <- 2;
while 2 * !p + 1 <= max do
  let next = t.arr.(!p) / 2 in
  (* on élimine les derniers multiples et on écrase le début de la liste chaînée *)
  if next <= t.max_arr then
    if t.arr.(next) < 0 then t.arr.(!p) <- - t.arr.(next)
    else begin
      t.arr.(!cnt) <- 2 * !p + 1;
      cnt := !cnt + 1;
      p := next end
  else begin
    t.arr.(!cnt) <- 2 * !p + 1;
    cnt := !cnt + 1;
    p := t.max_arr + 1 end
done;
Array.sub t.arr 0 !cnt (* extrait les nombres premiers écrits au début de t.arr *)
```

FIGURE 4 – Code OCaml correspondant à l'extraction des nombres premiers.

3 Une vérification formelle avec Why3

La preuve qui est décrite ici a été intégralement réalisée avec l'outil Why3 [2] et les démonstrateurs automatiques Alt-Ergo, CVC4, Z3, E et Vampire.

Spécification. La spécification de la fonction principale du crible, `euler_sieve`, est la suivante :

```
let euler_sieve (max: int63) : array63
  requires { max_int > max ≥ 3 }
  ensures  { ∀ i j. 0 ≤ i < j < result.length → result[i] < result[j] }
  ensures  { ∀ i. 0 ≤ i < result.length → 2 ≤ result[i] ≤ max }
  ensures  { ∀ i. 0 ≤ i < result.length → prime result[i] }
  ensures  { ∀ i. 2 ≤ i ≤ max → prime i →
                ∃ j. 0 ≤ j < result.length ∧ result[j] = i }
= ...
```

Le type `int63` est une modélisation Why3 des entiers OCaml qui sont des entiers 63 bits signés. De même, le type `array63` est une modélisation Why3 d'un tableau OCaml contenant des entiers OCaml. La précondition requiert que `max` ne soit pas égal au plus grand entier. En effet, le code a besoin de stocker la valeur `max+1` dans la dernière case du tableau pour signaler la fin.

La première postcondition énonce que les nombres du tableau renvoyé en résultat sont écrits dans l'ordre strictement croissant. La postcondition suivante énonce que les nombres renvoyés dans le tableau sont tous dans l'ensemble $\{2, ..., \texttt{max}\}$. Enfin, les deux dernières postconditions énoncent que tous les éléments du tableau renvoyé sont premiers et que tous les nombres premiers compris entre 2 et `max` sont dans le tableau. Le prédicat `prime` utilisé dans ces deux dernières postconditions provient de la bibliothèque `number.Prime` de Why3. Il a déjà été utilisé dans les preuves d'autres programmes manipulant des nombres premiers, comme le crible d'Ératosthène [4] ou celui de l'algorithme des nombres premiers de Knuth [9, 7].

Vérification. La preuve du crible d'Euler a été faite par raffinement [6], en deux temps. La première partie de la preuve a été faite, dans un module `EulerSieve`, à l'aide d'un type `t` représentant une structure abstraite. Ce type modélise la liste chaînée et le marquage, sans en connaître l'implémentation, de la manière suivante :

```
type t = private {
  mutable ghost nexts: seq int;
  mutable ghost marked: seq bool;
  max: int63;
}
```

Le champ `nexts` modélise la liste chaînée implémentée par le tableau `arr`. À la différence de ce dernier, cependant, tous les nombres y sont représentés et le marquage en est absent. Ce dernier est modélisé par le champ `marked`. Comme pour `nexts`, tous les nombres y sont représentés. Le type `seq` utilisé pour décrire la liste chaînée et le marquage est ici un type modélisant des séquences. Il s'agit de tableaux immuables fournis par la bibliothèque standard de Why3. Pour pouvoir les manipuler, il est donc nécessaire que les champs correspondants soient mutables. De plus, comme on peut le voir ci-dessus, les champs modélisant la liste chaînée et le marquage sont des champs fantômes [5]. Les champs fantômes n'existent que dans la preuve et pas dans le programme. Ils seront automatiquement supprimés par Why3 lors de l'extraction du programme, une fois démontré, vers du code OCaml. Ces champs fantômes sont ici très utiles puisqu'ils permettent de représenter des structures de données simples, ce qui a pour conséquence de faciliter la manipulation des structures de données dans la logique et, par conséquent, de faciliter la preuve.

Des fonctions permettant de modifier ces champs ont été déclarées. Elles sont définies de manière abstraites dans le module `EulerSieve`. Voici un exemple, extrait de la preuve, d'une telle fonction abstraite :

```
val set_mark (t: t) (i: int63) : unit
  requires { 0 ≤ i ≤ t.max }
  requires { mod i 2 = 1 }
  writes   { t.marked }
  ensures  { t.marked = (old t.marked)[i ← true] }
```

La deuxième partie de la preuve a consisté en le raffinement du type `t` précédemment défini. Ce raffinement se trouve dans le module `EulerSieveImpl` qui est une implémentation du module `EulerSieve`. La nouvelle définition du type `t` est la suivante :

```
type t = {
  mutable ghost nexts: seq int;
  mutable ghost marked: seq bool;
  arr: array63;
  max: int63;
  max_arr: int63
}
```

On observe l'ajout des champs `arr` et `max_arr` correspondant aux champs de mêmes noms définis dans la section 2. Il s'agit donc de l'implémentation de la liste chaînée et du marquage. Le champ `arr` est lié aux champs fantômes `nexts` et `marked` par des invariants de liaison dont voici un extrait :

```
invariant { ∀ i. 0 ≤ i ≤ max_arr →
                 Seq.get marked (2 * i + 1) ↔ arr[i] < 0 }
```

Les fonctions permettant de manipuler la liste chaînée et le marquage ont été ensuite implémentées dans le module `EulerSieveImpl`. Il a été nécessaire, après implémentation, de montrer que celles-ci satisfont les spécifications de leurs fonctions abstraites respectives et conservent les invariants du type `t`, tel que l'invariant de liaison évoqué ci-dessus.

Ainsi, la preuve du crible d'Euler a été faite uniquement dans la structure `t` abstraite, définie précédemment. Puis cette structure `t` a été réalisée avec les optimisations décrites en section 2 et nous avons montré que cette structure concrète respectait sa spécification et celle de la structure abstraite `t`.

La preuve a nécessité l'écriture de 775 lignes dans le langage WhyML : 517 lignes pour la spécification et 258 lignes de code. Au total, 840 buts ont été prouvés. Afin de démontrer le crible d'Euler, de nombreuses interactions ont été nécessaires pour aider les démonstrateurs automatiques. Parmi ces interactions, de nombreux lemmes ont été énoncés et prouvés. Des assertions ont également été écrites dans le programme. Enfin, des transformations logiques ont été appliquées à des buts à de nombreuses reprises. La répartition des buts démontrés par les différents démonstrateurs automatiques est la suivante :

démonstrateur	nombre de buts prouvés	temps maximum sur un but
Eprover 2.4	7	0,74 s
Vampire 4.4.0	6	8,22 s
Alt-Ergo 2.3.2	517	7,85 s
Alt-Ergo 2.0.0	89	8,28 s
Z3 4.8.6	137	1,84 s
CVC4 1.7	39	0,34 s
CVC4 1.6	45	4,05 s

4 Le code OCaml extrait de la preuve

Une fois la démonstration terminée, il est maintenant possible d'extraire le code OCaml du programme démontré. Ceci est fait par la commande `extract` de Why3. Le code OCaml obtenu après extraction est présenté à la figure 5. Seules quelques modifications de mise en page ont été effectuées entre le code extrait et le code présenté. Voici quelques mesures du temps de calcul et de la mémoire utilisée par le code extrait :

N	temps de calcul	mémoire utilisée
10^6	0,022 s	8 Mo
10^7	0,140 s	73 Mo
10^8	1,350 s	721 Mo
10^9	14,500 s	7 200 Mo

Afin de pouvoir évaluer l'efficacité de ce code, nous comparons maintenant les temps de calcul du code extrait de notre crible d'Euler avec des implémentations du crible d'Ératosthène et du crible d'Ératosthène segmenté [1]. Concernant ce dernier, la comparaison est fournie pour permettre de donner une idée de la différence d'efficacité, mais elle n'est pas totalement équitable. En effet, contrairement aux cribles d'Ératosthène et d'Euler, qui renvoie un tableau contenant tous les nombres premiers, le crible d'Ératosthène segmenté ne renvoie rien mais applique une fonction `int -> unit` donnée en argument sur tous les nombres premiers compris entre 2 et la limite.

1. Merci à Jean-Christophe Filliâtre pour son implémentation du crible d'Ératosthène segmenté.

```
type t = { arr: int array; max: int; max_arr: int }

let create max =
  let len_arr = (max - 1) / 2 + 1 in
  let arr = Array.make len_arr (-2) in
  for i = 1 to len_arr - 1 do
    arr.(i) <- if i = len_arr - 1 then max + 1 else 2 * i + 3
  done;
  { arr = arr; max = max; max_arr = (max - 1) / 2 }

let set_next t i v = t.arr.(i / 2) <- v
let get_next t i = if t.arr.(i / 2) < 0 then - t.arr.(i / 2) else t.arr.(i / 2)
let set_mark t i = if t.arr.(i / 2) >= 0 then t.arr.(i / 2) <- - t.arr.(i / 2)
let get_mark t i = t.arr.(i / 2) < 0
let get_max t = t.max

let remove_products t n =
  let d = get_max t / n in
  let rec loop (p: int) : unit =
    let next = get_next t p in
    if 0 <= next && next <= get_max t then begin
      if next <= d then begin
        set_mark t (n * next);
        if get_mark t next then begin
          set_next t p (get_next t next); loop p
        end else loop next
      end end in
  set_mark t (n * n); loop n

let euler_sieve max =
  let t = create max in
  let rec loop n =
    remove_products t n;
    let nn = get_next t n in
    if nn <= max / nn then loop nn in
  if max >= 9 then loop 3;
  let cnt = ref 1 in
  let p = ref 1 in t.arr.(0) <- 2;
  while 2 * !p + 1 <= max do
    let next = t.arr.(!p) / 2 in
    if next <= t.max_arr then
      if t.arr.(next) < 0 then t.arr.(!p) <- - t.arr.(next)
      else begin t.arr.(!cnt) <- 2 * !p + 1; cnt := !cnt + 1; p := next end
    else begin t.arr.(!cnt) <- 2 * !p + 1; cnt := !cnt + 1; p := t.max_arr + 1 end
  done;
  Array.sub t.arr 0 !cnt
```

FIGURE 5 – Le code extrait en OCaml du crible d'Euler.

N	crible d'Ératosthène	crible d'Euler	crible d'Ératosthène segmenté
10^6	0,033 s	0,022 s	0,013 s
10^7	0,186 s	0,140 s	0,039 s
10^8	1,970 s	1,350 s	0,312 s
10^9	21,780 s	14,500 s	3,390 s

On peut observer ici que le code du crible d'Euler est plus rapide que l'implémentation du crible d'Ératosthène que nous utilisons. Cependant, les performances sont encore éloignées de celles du crible d'Ératosthène segmenté, qui tire parti de la rapidité offerte par la mémoire cache du processeur.

5 Conclusion

Dans cet article, nous avons présenté une implémentation du crible d'Euler. Nous en avons montré la spécification et nous avons prouvé que l'implémentation fournie respectait cette spécification. La preuve a été réalisée par raffinement, à l'aide de différents outils fournis par Why3. Le code extrait de la preuve par Why3 possède de bonnes performances, meilleures que celles fournies par un crible d'Ératosthène. Cependant, elles sont encore loin des performances du crible d'Ératosthène segmenté, bien que celui-ci ne réponde pas exactement au même problème. Il serait ainsi intéressant, dans le même but que pour le crible d'Euler, de vérifier formellement une implémentation du crible d'Ératosthène segmenté. Une autre perspective intéressante serait de prouver formellement que l'implémentation du crible d'Euler que nous avons présentée est bien de complexité linéaire, par exemple à l'aide de crédits temps [3].

Remerciements. Je remercie chaleureusement Jean-Christophe Filliâtre pour le stage effectué sous sa direction, ainsi que pour le temps qu'il m'a consacré. Merci aussi pour toutes les choses passionnantes qu'il m'a fait découvrir dans les domaines des méthodes formelles, de la compilation et de la programmation en particulier, ainsi que pour ses nombreux conseils et encouragements durant la rédaction de cet article et pour sa relecture. Je remercie également tous ceux qui ont relu cet article pour leurs corrections et conseils.

Références

[1] Preuve en Why3 du crible d'Euler. https://github.com/aistun/EulerSieve/tree/master/.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform. http://why3.lri.fr/.

[3] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3) :331–365, March 2019.

[4] Martin Clochard. Sieve of Eratosthenes. http://toccata.lri.fr/gallery/sieve.en.html.

[5] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

[6] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Lecture Notes in Computer Science, Rhodes, Greece, October 2020. http://why3.lri.fr/isola-2020/.

[7] Jean-Christophe Filliâtre. Knuth's prime numbers. http://toccata.lri.fr/gallery/knuth_prime_numbers.en.html.

[8] David Gries and Jayadev Misra. A linear sieve algorithm for finding prime numbers. *Commun. ACM*, 21(12) :999–1003, December 1978.

[9] Donald E. Knuth. *The Art of Computer Programming*, volume 1, page 147. Addison Wesley Professional, 1997.