

# Raisonner à propos du temps en logique de séparation

François Pottier

en collaboration avec A. Charguéraud, A. Guéneau, G. Mével, J.-H. Jourdan



1<sup>er</sup> avril 2021

*En r'tard, en r'tard  
J'ai rendez-vous que'que part  
Je n'ai pas le temps de dire au revoir  
Je suis en r'tard, en r'tard*

# Quelles propriétés des programmes vérifions-nous ?

Quand on parle de vérifier un programme, on pense souvent à garantir :

- la *sûreté*, l'absence d'erreurs graves pendant l'exécution ;  
*overflows, divisions par zéro, accès en dehors des bornes...*
- la *correction fonctionnelle*, le fait de produire un résultat correct ;
- la *terminaison*.

# Quelles propriétés des programmes vérifions-nous ?

Toutefois, on peut aussi souhaiter garantir des propriétés de *sécurité*, ou encore contrôler l'utilisation de *ressources* telles que :

- la mémoire vive,
- la mémoire de masse (les disques durs et disques SSD),
- la communication via un réseau,
- la consommation électrique,
- le *temps*.

# Comment raisonner à propos du temps ?

Nous avons déjà mentionné la *non-terminaison*,  
le cas le plus évident d'abus de la ressource "temps".

- C'est (habituellement) un comportement indésirable.
- Il est relativement facile à écarter en pratique.
  - *on associe un variant aux boucles et aux fonctions récursives*

# Comment raisonner à propos du temps ?

Terminer, c'est bien, mais terminer quand ?

Peut-on garantir la terminaison *dans un délai* explicitement borné ?



# Comment raisonner à propos du temps ?

Terminer, c'est bien, mais terminer quand ?

Peut-on garantir la terminaison *dans un délai* explicitement borné ?



On imagine assez facilement les attraites et difficultés d'une telle idée...

Parmi les attraits,

- Éliminer une catégorie *d'erreurs de programmation*.
  - *on commet parfois une erreur qui compromet la complexité mais pas la correction.*
  - *dommage, car si on a choisi un algorithme plus évolué, c'est bien pour gagner en performance !*
- Éliminer une catégorie de *failles de sécurité*.
  - *p.ex. attaques de déni de service contre des tables de hachage*

On ne maîtrise pas le *temps réel*.

On peut compter les opérations, ou les cycles.

C'est fastidieux. De plus, le compte exact dépend

- du *langage* de programmation choisi,
- du *compilateur* utilisé,
- du *processeur* utilisé,
- des aléas du système d'exploitation, du système mémoire, etc.

Nous ne parlerons donc pas ici d'analyse du temps réel (*WCET*).

On veut une façon plus abstraite de mesurer l'efficacité d'un programme.

Compter les opérations *à une constante près*,  
et laisser cette constante non spécifiée.

- Publier  $O(n)$  au lieu de  $13n + 237$ .
- La constante cachée peut dépendre du code, pas des données.

Cette information incomplète n'empêche pas les comparaisons :

- $O(n \log n)$  est plus rapide que  $O(n^2)$  pour  $n$  *assez grand*.

Ce style de spécification est *modulaire* et *robuste*.

- Remplacer  $13n + 237$  par  $9n + 323$  n'affecte pas les utilisateurs.

Compter *seulement certaines opérations* représentatives.

Dans un tri, on pourrait ne compter que les comparaisons.

On peut toujours se contenter de compter les *appels de fonction* et les *entrées dans le corps d'une boucle*.

Dans la suite, je supposerai qu'on a inséré dans le code des *tick()* et je ne compterai que ces pseudo-instructions.

Le *placement* des *ticks* détermine ce que l'on compte ;  
le *comptage* des *ticks* est indépendant de leur placement.

Hopcroft (1986) rappelle l'état de l'art dans les années 1960 :

*Research on algorithms [was] very unsatisfying. A researcher would publish an algorithm [...] along with execution times for a small set of sample problems. [...] Later, a second researcher would give an improved algorithm along with execution times for the same set of sample problems. The new algorithm would invariably be faster, since in the intervening years, both computer performance and programming languages had improved.*

Il raconte comment l'idée de complexité asymptotique a été accueillie :

*The idea met with much resistance. People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true, however, since as computers become faster, the size of the attempted problems becomes larger.*

Skiena (1997) confirme cela : « I was a Victim of Moore's Law ».

*Over the years, my users started asking why it took so long to do a modest-sized graph computation. The mystery wasn't that my program was slow. The question was : why did it take so many years for people to figure this out ?*

Plus les machines s'améliorent, plus les algos inefficaces semblent lents !

Tarjan, co-récepteur du prix Turing (1986), écrit :

*As a measure of computational efficiency, we settled on the worst-case time, as a function of the input size, of an algorithm [...]. We chose to ignore constant factors in running time, so that our measure could be independent of any machine model and of the details of any algorithm implementation. An algorithm efficient by this measure tends to be efficient in practice. This measure is also analytically tractable, which meant that we would be able to actually derive interesting results about it.*

Tarjan ajoute :

*What is important in most applications is the time taken by **an entire sequence of operations** rather than by a single operation.*

C'est la complexité *amortie* ;  
nous en reparlerons plus loin.



Simple problème de comptage, donc ? Oui, mais :

- à partir d'un véritable code source, pas d'un pseudo-code
- dans le cadre d'un système de vérification de programmes
  - une propriété de *correction* peut aider la preuve de complexité,
  - une propriété de *possession* peut aider la preuve de complexité,
  - un argument de complexité peut aider la preuve de correction !

Un spectre de questions, mathématiques / ingénierie / informatique :

- comprendre  $O(\dots)$  en présence de plusieurs paramètres
- gérer les *quantificateurs*  $\exists$  *cachés* dans les  $O(\dots)$
- automatiser (autant que possible) la *synthèse* de fonctions de coût
- écrire des spécifications *modulaires* : précises, *pas trop* précises
- autoriser arguments de correction et de complexité à *interagir*
- concevoir et justifier une *logique de programmes* adaptée

Je me focalise aujourd'hui principalement sur les deux derniers points.

J'aborderai aujourd'hui

- le comptage *global* du temps en logique de Hoare
- le comptage *local* du temps en logique de séparation  
les *crédits* et l'analyse amortie
- l'analyse amortie de programmes *paresseux*  
les *débites* et leur lien avec les crédits
- les vertus de la *lenteur*  
les *reçus*

- 1 Un temps global en logique de Hoare
- 2 Un temps local en logique de séparation : les crédits
- 3 Un temps pour la paresse : les débits
- 4 Un souvenir du temps passé : les reçus
- 5 Conclusion

La logique de Hoare garantit une propriété de *l'état final* atteint lors d'une exécution.

Si l'état comprend un *historique*, alors elle peut garantir une propriété *d'une exécution toute entière*.

Si l'état comprend une *horloge*, alors elle peut garantir que l'exécution *ne dure pas plus* d'un certain temps.

Ajoutons à l'état du programme une horloge  $H$ , un compteur entier.

Considérons que  $tick()$  décrémente  $H$  *et peut planter* :

$$tick() = \text{if } H > 0 \text{ then } H := H - 1 \text{ else } crash()$$

Supposons que  $H$  n'est pas modifié en dehors de  $tick()$ .

Alors, vérifier qu'un programme est *sûr* (ne plante pas)

*suffit* à garantir qu'il n'exécutera pas plus de  $H_0$  instructions  $tick()$ ,  
où  $H_0$  est la valeur initiale de  $H$ .

La logique de Hoare permet donc de *raisonner à propos du temps!*

De plus, si nous vérifions que l'instruction *crash()* n'est jamais atteinte, alors les appels à *tick()* n'ont aucun effet et peuvent être supprimés.

Bien que l'horloge *H* ne soit pas *a priori* une variable fantôme, elle se révèle *a posteriori* effaçable. Elle n'est qu'un *artifice de raisonnement*.

Un lemme fondamental est la spécification de *tick()* :

$$\forall h. \{H = h \wedge h > 0\} \text{ tick}() \{H = h - 1\}$$

On peut montrer en logique de Hoare que ce triplet est valide, et ce, même en l'absence d'un triplet qui décrit *crash()*.

La précondition exige de *montrer qu'il reste du temps*.

La postcondition reflète le fait que le temps restant décroît.

## Quelques spécifications typiques

On raisonne ensuite à propos de programmes *“tiqués”*,  
c'est à dire dans lesquels on a placé des instructions *tick()*.

À une fonction qui calcule la somme des éléments d'un tableau,  
on pourrait donner cette spécification :

$$\begin{aligned} & \{|A| = n \wedge H = h \wedge h \geq n + 1\} \\ & \quad S := \text{sum}(A) \\ & \{S = \sum_{i=0}^{i < n} A[i] \wedge H = h - (n + 1)\} \end{aligned}$$

Elle indique que ce calcul consomme *exactement*  $n + 1$  unités de temps.

Spécification ordinaire et spécification de la complexité sont *combinées*.

- *elles sont vérifiées ensemble*
- *la preuve de correction peut aider la preuve de complexité*

## Quelques spécifications typiques

En général, il sera difficile de décrire exactement le temps consommé ; on se contentera d'exprimer une *borne supérieure*.

Par exemple, à une fonction *lsearch* qui cherche un élément dans un tableau, on pourrait attribuer cette spécification :

$$\begin{aligned} &\{|A| = n \wedge H = h \wedge h \geq n + 1\} \\ &\quad I := \text{lsearch}(A, X) \\ &\quad \{\dots \wedge H \geq h - (n + 1)\} \end{aligned}$$

Elle indique que ce calcul consomme *au plus*  $n + 1$  unités de temps.

## Théorème (Sûreté)

*Si  $\{H = n\} p \{True\}$  est dérivable, alors une exécution quelconque du programme  $p$  ne contient pas plus de  $n$  instructions  $tick()$ .*

Ce théorème ne parle pas directement du *temps* d'exécution de  $p$ .

– En réalité, seul, il n'en garantit même pas la terminaison !

Mais si on a habilement placé les instructions  $tick()$  dans  $p$ , alors *le nombre total* d'instructions exécutées et *le nombre de  $tick()$*  exécutés ne peuvent différer que d'un facteur constant.

Crary et Weirich (2000) présentent une approche fondée sur un compteur global, sous la forme d'un système de types.

Dockins et Hobor (2012) proposent une logique analogue, en présence de fonctions d'ordre supérieur.

Haslbeck et Nipkow (2018) définissent et comparent formellement plusieurs logiques capables de raisonner à propos du temps.

- ① Un temps global en logique de Hoare
- ② **Un temps local en logique de séparation : les crédits**
  - Principe
  - Illustration
- ③ Un temps pour la paresse : les débits
- ④ Un souvenir du temps passé : les reçus
- ⑤ Conclusion

La logique de séparation permet de

- ne mentionner dans une spécification *que ce dont on a besoin*,
- penser en termes de *possession* et de *transferts* de possession.

Peut-on appliquer cela au temps ?

La logique de séparation permet de

- ne mentionner dans une spécification *que ce dont on a besoin*,
- penser en termes de *possession* et de *transferts* de possession.

Peut-on appliquer cela au temps ?

L'idée est de ne plus mentionner l'horloge globale  $H$   
mais de *répartir* la possession du temps restant à dépenser.

② Un temps local en logique de séparation : les crédits

Principe

Illustration

Un crédit-temps, noté  $\$1$ , est une assertion.

– comme l'assertion  $I \mapsto v$ , par exemple

Elle représente *la possession d'une unité*  
de la valeur actuelle de l'horloge  $H$ .

On maintient *l'invariant* que si la variable  $H$  contient la valeur  $h$ , alors il y a au plus  $h$  crédits-temps en circulation.

Un crédit-temps, noté  $\$1$ , est une assertion.

– comme l'assertion  $I \mapsto v$ , par exemple

Elle représente *la possession d'une unité* de la valeur actuelle de l'horloge  $H$ .

On maintient *l'invariant* que si la variable  $H$  contient la valeur  $h$ , alors il y a au plus  $h$  crédits-temps en circulation.

Une analogie monétaire utile :

- $h$  lingots d'or sont stockés à la Banque de France ;
- au plus  $h$  billets de banque sont en circulation ;
- chaque billet est une preuve d'existence d'un lingot à la Banque.

# Propriété fondamentale des crédits-temps

On peut établir pour *tick()* la spécification suivante :

$$\{\$1\} \text{ tick()} \{ True \}$$

Ce triplet affirme que *tick()* *consomme* un crédit.

Un crédit est donc une *permission* à dépenser une unité de temps.

# Propriété fondamentale des crédits-temps

On peut établir pour  $tick()$  la spécification suivante :

$$\{\$1\} tick() \{ True \}$$

Ce triplet affirme que  $tick()$  *consomme* un crédit.

Rappelons que  $tick()$  décrémente  $H$  et peut planter :

$$tick() = \text{if } H > 0 \text{ then } H := H - 1 \text{ else } crash()$$

L'assertion  $\$1$  permet de montrer que  $H > 0$  doit être vrai.

La décrémentation de  $H$  préserve l'invariant, si on abandonne  $\$1$ .

- de un billet et un lingot, on passe à zéro billet et zéro lingot
- l'invariant “#billets  $\leq$  #lingots” reste satisfait

On peut établir pour  $tick()$  la spécification suivante :

$$\{\$1\} tick() \{ True \}$$

Ce triplet affirme que  $tick()$  *consomme* un crédit.

On peut le comparer au précédent, exprimé en logique de Hoare :

$$\forall h. \{ H = h \wedge h > 0 \} tick() \{ H = h - 1 \}$$

On ne fait plus référence à  $H$ . On a *décentralisé* le comptage du temps.

## Autres propriétés des crédits-temps

Les crédits-temps se comportent comme des billets de banque.

Un crédit est *affine* (jetable), mais *pas duplicable*.

– jeter un crédit revient à sur-approximer un coût

Un crédit ne peut pas être créé à partir de rien.

Les crédits peuvent être assemblés et séparés :

$$\begin{aligned} \$(m + n) &\equiv m\$ \star \$n \\ \$0 &\equiv True \end{aligned}$$

La règle d'encadrement donne alors pour *tick()* la spécification :

$$\{\$(n + 1)\} \text{ tick() } \{\$n\}$$

Le théorème qui affirme la sûreté de la logique est inchangé :

### Théorème (Sûreté)

*Si  $\{n\} p \{True\}$  est dérivable, alors une exécution quelconque du programme  $p$  ne contient pas plus de  $n$  instructions `tick()`.*

La logique de séparation avec crédits permet une analyse *au pire cas*.

À la fonction *lsearch*, on pourrait attribuer cette spécification :

$$\{ \text{isArray}(a, xs) \star |xs| = n \star \$(n + 1) \}$$
$$l := \text{lsearch}(a, x)$$
$$\{ \text{isArray}(a, xs) \star \dots \}$$

Elle indique que *lsearch* *consomme*  $n + 1$  crédits-temps, où  $n$  est la longueur du tableau  $a$ .

Cela *ne garantit pas* que *lsearch*( $a, x$ ) exécute au plus  $n + 1$  *ticks*.

- Cette garantie n'est donnée que pour les programmes complets.

De façon générale, face à une spécification de cette forme :

$$\{P \star \$n\} c \{Q\}$$

on *ne peut pas* conclure que la commande  $c$  exécute au plus  $n$  *tick()*,  
parce que l'assertion  $P$  pourrait donner accès à d'autres crédits-temps,  
en plus des  $n$  crédits visibles ici.

C'est précisément cela qui va permettre l'analyse *amortie*.

② Un temps local en logique de séparation : les crédits

Principe

Illustration

Prenons l'exemple d'une *file d'attente*  
implémentée par deux listes chaînées *front* et *rear*.

Cette structure est *immuable* donc persistante :  
insérer ou extraire un élément produit une nouvelle file.

```
let put (Q (front, rear)) x =
  Q (front, x :: rear)
```

*– insert into rear list*

```
let get (Q (front, rear)) =
  match front with
  | x :: front →
    Some (x, Q (front, rear))
  | [] →
    match revappend rear [] with
    | x :: rear →
      Some (x, Q (rear, []))
    | [] →
      None
```

*– extract out of front list*

*– if front list is empty,  
– reverse rear list (costly)*

*– and make it the front list*

*– both lists are empty ; return nothing*

En logique de séparation ordinaire, pour commencer.

On publie l'existence d'un prédicat abstrait  $isQueue(q, xs)$ .

On publie les spécifications :

$$\begin{array}{ll}
 \{ isQueue(q, xs) \} & \{ isQueue(q, []) \} \\
 \text{put } q \ x & \text{get } q \\
 \{ q'. isQueue(q', xs ++ [x]) \} & \{ r. r = None \}
 \end{array}$$

$$\left\{ \begin{array}{l}
 isQueue(q, x :: xs) \\
 \text{get } q \\
 r. \exists q'. r = Some(x, q') \star \\
 isQueue(q', xs)
 \end{array} \right\}$$

Pour vérifier cette structure de données, il faut définir son *invariant* interne, qui pourrait ressembler à ceci :

$$\begin{aligned} isQueue(q, xs) = \\ \exists front, rear. q = Q(front, rear) \wedge xs = front ++ rev(rear) \end{aligned}$$

Cet invariant est pur donc *duplicable* :

$$isQueue(q, xs) \vdash isQueue(q, xs) \star isQueue(q, xs)$$

Deux acteurs indépendants peuvent utiliser une même file.

# Spécification de la file d'attente

En logique de séparation avec crédits-temps, ensuite.

Je suppose qu'un *tick()* est inséré au début de chaque fonction.

On adapte les spécifications :

$$\begin{array}{ll} \{ isQueue(q, xs) \star \$1 \} & \{ isQueue(q, []) \star \$1 \} \\ \text{put } q \ x & \text{get } q \\ \{ q'. isQueue(q', xs ++ [x]) \} & \{ r. r = None \} \end{array}$$

$$\begin{array}{l} \left\{ \begin{array}{l} isQueue(q, x :: xs) \star \\ |x :: xs| = n \star \$(n+1) \end{array} \right\} \\ \text{get } q \\ \left\{ \begin{array}{l} r. \exists q'. r = Some(x, q') \star \\ isQueue(q', xs) \end{array} \right\} \end{array}$$

On adapte la preuve, sans modifier *isQueue*.

À chaque *tick()*, on dépense un crédit.

Toutefois, cette analyse peut sembler très *pessimiste* :  
chaque appel à *get* exige  $n + 1$  crédits,  
bien qu'en réalité le pire cas ne se produise que rarement.

Une analyse *amortie* est plus précise...

Le principe de l'analyse amortie est bien connu (Tarjan, 1985) :

- *augmenter le coût apparent* de certaines opérations,
- pour pouvoir *économiser*, et ainsi
- *piocher dans le bas de laine* en cas de coup dur.

Dans le cas de la file d'attente, l'idée est de :

- prétendre que chaque *put* coûte non pas 1 mais *2 crédits*,
- pour *disposer en permanence de  $n$  crédits*, où  $n = |\text{rear}|$ , et ainsi
- couvrir le coût de l'appel à *revappend* quand il a lieu.

Pour exprimer cela, l'invariant doit *posséder* des crédits.

– *on ne pouvait pas exprimer cela en logique de Hoare*

L'invariant interne devient donc :

$$isQueue(q, xs) = \exists front, rear. q = Q(front, rear) \wedge xs = front ++ rev(rear) \star \$|rear|$$

Après avoir adapté la preuve, on obtient une spécification différente :

$$\begin{array}{cc} \{ isQueue(q, xs) \star \$2 \} & \{ isQueue(q, []) \star \$1 \} \\ \text{put } q \ x & \text{get } q \\ \{ q'. isQueue(q', xs ++ [x]) \} & \{ r. isQueue(q, []) \star r = None \} \end{array}$$

$$\begin{array}{c} \{ isQueue(q, x :: xs) \star \$1 \} \\ \text{get } q \\ \{ r. \exists q'. r = Some(x, q') \star isQueue(q', xs) \} \end{array}$$

On a placé des crédits-temps dans le prédicat abstrait *isQueue*.

Même si *get* consomme *visiblement* seulement 1 crédit,

– son *coût amorti* est 1

son coût réel peut être supérieur.

$$\{isQueue(q, xs) \star \$1\} get\ q \{...\}$$

L'analyse amortie est *incorrecte* si on utilise *deux fois* une structure :

```
get q1 >>= fun (x, q2) → - spend my savings!
```

```
let q3 = put q2 y in
```

```
get q1 >>= fun (z, q4) → - spend my savings? they are gone already!
```

...

Ce code ne pourra pas être vérifié car *isQueue n'est plus duplicable* mais affine.

La présence de crédits-temps dans une structure de données impose de *renoncer au partage* de cette structure.

## Deux spécifications incomparables

Nous avons donné *deux spécifications incomparables* de la file d'attente.

- Dans la première, *isQueue* est duplicable, mais *get* coûte  $O(n)$ .
- Dans la seconde, *get* coûte \$1 seulement, mais *isQueue* est affine.

Il peut exister *plusieurs présentations de la complexité* d'un algorithme, et différents clients ont besoin de différentes analyses.

C'est un problème ouvert potentiellement difficile.

Atkey (2011) et Pilkiewicz et Pottier (2011) introduisent l'idée des crédits-temps.

Nipkow (2015) vérifie l'analyse amortie de diverses structures de données, sans crédits-temps.

Charguéraud et Pottier (2017), Zhan et Haslbeck (2018), Guéneau et al. (2019), Haslbeck et Lammich (2021) exploitent les crédits-temps pour vérifier divers algorithmes.

Guéneau et al. (2019) proposent les crédits-temps *négatifs*.

Mével et al. (2019) définissent les crédits-temps à l'aide de l'état fantôme et des invariants d'Iris.

- ① Un temps global en logique de Hoare
- ② Un temps local en logique de séparation : les crédits
- ③ Un temps pour la paresse : les débits**
- ④ Un souvenir du temps passé : les reçus
- ⑤ Conclusion

L'analyse amortie de la file d'attente est *plus précise*  
que l'analyse pessimiste ordinaire  
mais impose de *renoncer au partage*.

Le problème provient du danger qu'une opération coûteuse,  
le renversement de la liste *rear*,  
soit effectué deux fois.

Pourrait-on éviter cela ?

L'analyse amortie de la file d'attente est *plus précise*  
que l'analyse pessimiste ordinaire  
mais impose de *renoncer au partage*.

Le problème provient du danger qu'une opération coûteuse,  
le renversement de la liste *rear*,  
soit effectué deux fois.

Pourrait-on éviter cela ?

Pourrait-on *mémoiser* les calculs coûteux pour éviter de les répéter ?

Une *suspension* est une structure de données très simple qui représente soit un calcul en attente, soit le résultat de ce calcul.

En OCaml, on la trouve dans la bibliothèque *Lazy* :

```
type 'a t  
val fromfun : (unit → 'a) → 'a t  
val force : 'a t → 'a
```

Une suspension construite par *fromfun f* peut être “forcée” zéro, une, ou plusieurs fois, mais le calcul  $f()$  aura lieu une fois au plus.

Utiliser des suspensions, c'est bien, mais cela ne suffit pas.

Il est essentiel de les créer *suffisamment longtemps avant* que leur valeur ne soit exigée, pour que leur résultat soit *réutilisé suffisamment de fois*.

Okasaki (1999) propose de *prévoir* – mais non pas *effectuer* – le renversement de la liste *rear* dès que celle-ci devient assez longue – par exemple, plus longue que la liste *front*. Il impose donc  $|rear| \leq |front|$ .

Il baptise cette structure *file du banquier* ou *banker's queue*.

```
type 'a stream =  
  'a cell Lazy.t   – computation is suspended
```

```
and 'a cell =  
  | Nil  
  | Cons of 'a × 'a stream
```

```
type 'a queue = {  
  lenf : int;      – length of front sequence  
  f : 'a stream;  – lazy front sequence, computed on demand  
  lenr : int;      – length of rear sequence  
  r : 'a list;     – ordinary rear sequence, stored in memory  
}                 – invariant : lenf ≥ lenr
```

```

let put q x =
  check { q with           - rebalance
    lenr = q.lenr + 1;
    r = x :: q.r;           - insert into rear list
  }

let get q =
  match extract q.f with - extract out of front sequence (forcing a thunk)
  | None →                 - if front list is empty...
    None                   - ...then rear list is empty too
  | Some (x, f) →
    Some (x,
      check { q with     - rebalance
        f = f;
        lenf = q.lenf - 1;
      })

```

La fonction *check* renverse la liste *rear* si elle est devenue trop longue.

```

let check ({ lenf = lenf ; f = f ; lenr = lenr ; r = r } as q) =
  if lenf ≥ lenr then
    q                                - invariant holds; nothing to do
  else {                             - we have lenf + 1 = lenr
    lenf = lenf + lenr ;
    f = f ++ rev r ;                - re-establish invariant
    lenr = 0 ;
    r = [];
  }
  
```

La suspension *rev r* est coûteuse, mais ne sera forcée que lorsque tous les éléments de *f* auront été extraits, donc son coût par élément extrait est constant.

La fonction *check* renverse la liste *rear* si elle est devenue trop longue.

```
let check ({ lenf = lenf ; f = f ; lenr = lenr ; r = r } as q) =  
  if lenf ≥ lenr then  
    q                                - invariant holds; nothing to do  
  else {                             - we have lenf + 1 = lenr  
    lenf = lenf + lenr ;  
    f = f ++ rev r ;                 - re-establish invariant  
    lenr = 0 ;  
    r = [] ;  
  }
```

La suspension *rev r* est coûteuse, mais ne sera forcée que lorsque tous les éléments de *f* auront été extraits, donc son coût par élément extrait est constant.

*Vous y croyez ? Vous ne devriez pas. Demandez une preuve !*

Okasaki (1999) explique comment analyser informellement la complexité *amortie* de ces structures paresseuses.

Il y parvient, bien que les suspensions soient *partagées*,  
et sans imposer aucune restriction sur ce partage.

Okasaki (1999) explique comment analyser informellement la complexité *amortie* de ces structures paresseuses.

Il y parvient, bien que les suspensions soient *partagées*,  
et sans imposer aucune restriction sur ce partage.

L'idée-clef est de ne pas raisonner en termes de crédits, mais de *débits*.

- Si  $isQueue(q, xs)$  contient des crédits, alors  $isQueue(q, xs) \not\vdash isQueue(q, xs) \star isQueue(q, xs)$ , ce qui nous interdit de partager la file d'attente.
- Si un débit représente une *dette*, alors on peut dupliquer un débit : cela nous conduit à *augmenter* notre dette, donc à *sur-approximer* le coût réel du calcul analysé.

Okasaki associe un *débit*, une dette, à chaque suspension.

Il autorise *un paiement en plusieurs fois*.

Le débit est donc ce qui *reste à payer*  
avant d'être autorisé à forcer cette suspension.

Je dirai parfois que c'est le "coût" de cette suspension,  
mais ce n'est qu'un *coût résiduel* virtuel,  
inférieur au coût réel du calcul suspendu.

Le raisonnement d'Okasaki est informel.

Danielsson (2008) formalise les idées d'Okasaki sous forme de typage.

$Thunk\ n\ a$  est le type d'une suspension à laquelle sont associés  $n$  débits.

Il propose les opérations suivantes :

$return : a \rightarrow Thunk\ 0\ a$

$tick : Thunk\ n\ a \rightarrow Thunk\ (n + 1)\ a$

$bind : Thunk\ m\ a \rightarrow (a \rightarrow Thunk\ n\ b) \rightarrow Thunk\ (m + n)\ b$

$wait : Thunk\ m\ a \rightarrow Thunk\ (m + n)\ a$

$pay : Thunk\ n\ a \rightarrow Thunk\ m\ (Thunk\ (n - m)\ a)$

Il applique ces règles à la file du banquier (entre autres).

Peut-on traduire les idées d'Okasaki et Danielsson  
en logique de séparation ?

Peut-on *expliquer les débits* en termes de crédits-temps ?

Si oui, on obtiendra un système unifié où  
crédits (pour les calculs ordinaires)  
et débits (pour les calculs suspendus)  
cohabitent.

Que nous faut-il ?

Un prédicat *isThink*  $t$   $n$   $\phi$

signifiant que  $t$  est une suspension

à laquelle est associée une dette  $n$

et dont la valeur  $v$  satisfait ou satisfera  $\phi(v)$ .

Dans la suite, je parlerai de suspension de coût  $n$ .

On veut que ce prédicat soit *duplicable*, comme chez Danielsson :

$$isThink\ t\ n\ \phi \vdash isThink\ t\ n\ \phi \star isThink\ t\ n\ \phi$$

On souhaite pouvoir *créer* une suspension :

$$\{\$1 \star (\$n \multimap wp\ f()\ \phi)\}$$

*fromfun f*

$$\{\lambda t. isThunk\ t\ n\ \phi\}$$

Si l'appel de fonction  $f()$  exige  $n$  crédits,  
et si on dépense un crédit immédiatement,  
on obtient une suspension de coût  $n$ .

On peut lire l'implication  $\$n \multimap wp\ f()\ \phi$   
comme une forme affaiblie du triplet  $\{\$n\}\ f()\ \{\phi\}$ .  
Elle n'autorise qu'un appel à  $f()$  au plus.

On peut *forcer* une suspension si aucune dette ne lui est associée :

$\phi$  est duplicable  $\rightarrow \{ \$1 \star isThink\ t\ 0\ \phi \}$  force  $t\ \{ \phi \}$

Cette règle exige que la postcondition  $\phi$  soit duplicable.

En effet, si une suspension est forcée plusieurs fois, son résultat est *copié*.

On souhaite pouvoir *surestimer* une dette :

$$n_1 \leq n_2 \star \text{isThunk } t \ n_1 \ \phi \vdash \text{isThunk } t \ n_2 \ \phi$$

On souhaite une forme de règle de *conséquence* :

$$(\forall v, \phi_1(v) \multimap \phi_2(v)) \star \text{isThunk } t \ n \ \phi_1 \vdash \text{isThunk } t \ n \ \phi_2$$

De cette dernière, on peut déduire une règle de *transmission* :

$$\text{isThunk } t \ n \ \phi \star P \vdash \text{isThunk } t \ n \ (\lambda v. \phi(v) \star P)$$

En remplaçant  $P$  par  $\$k$ , on peut transmettre des crédits vers l'avenir.

On *retarde* ces crédits : il seront disponibles plus tard.

On souhaite pouvoir *payer* pour réduire un coût apparent :

$$\$k \star \text{isThink } t \ n \ \phi \vdash \text{isThink } t \ (n - k) \ \phi$$

Ainsi, des crédits existants servent à réduire une dette.

Inversement, on peut *emprunter* pour faire apparaître des crédits :

$$\text{isThink } t \ n \ \phi \vdash \text{isThink } t \ (n + k) \ (\lambda v. \phi(v) \star \$k)$$

Notre dette immédiate augmente, mais un gain futur en découle.

Tout ceci n'est que *jeu comptable* :

ces règles de raisonnement n'ont aucun effet à l'exécution ;  
c'est la même suspension qui est associée à différents coûts virtuels.

Par composition des deux règles précédentes, on obtient :

$$\begin{aligned} & isThunk\ t\ m\ (\lambda u. isThunk\ u\ n\ \phi) \\ \vdash & isThunk\ t\ (m + k)\ (\lambda u. \$k \star isThunk\ u\ n\ \phi) \\ \vdash & isThunk\ t\ (m + k)\ (\lambda u. isThunk\ u\ (n - k)\ \phi) \end{aligned}$$

Quand deux suspensions sont imbriquées, le coût de la seconde peut être (totalement ou partiellement) *déplacé* et imputé à la première.

On *anticipe* ainsi une dette : il faudra payer plus tôt.

Cette règle d'anticipation sert dans l'analyse de la file du banquier.

On construit une liste paresseuse  $f ++ rev r$ .

Les séquences  $f$  et  $r$  ont approximativement même longueur  $n$ .

- $f$  est une liste paresseuse, donc une chaîne de suspensions imbriquées, *toutes de coût nul* en ce point du code.
- $rev r$  est une suspension de coût  $n$ .

Par anticipation, ce coût peut être *étalé* sur toute la liste paresseuse : on obtient *un coût constant* pour chaque suspension de  $f ++ rev r$ .

# Spécification de la file du banquier

Pour résumer, on obtient la spécification suivante :

*isQueue(q, xs) est duplicable*

$$\begin{array}{c} \{isQueue(q, xs) \star \$P\} \\ \text{put } q \ x \\ \{q'. isQueue(q', xs ++ [x])\} \end{array}$$

$$\begin{array}{cc} \{isQueue(q, x :: xs) \star \$G\} & \{isQueue(q, []) \star \$G\} \\ \text{get } q & \text{get } q \\ \{r. \exists q'. r = \text{Some } (x, q') \star isQueue(q', xs)\} & \{r. r = \text{None}\} \end{array}$$

où  $P$  et  $G$  sont des constantes.

Cette spécification est *plus forte* que les deux précédentes.

Jusqu'ici, j'ai *postulé* les règles que doit satisfaire *isThunk*.

On voudrait *vérifier*,

    dans une logique de séparation assez puissante,  
qu'une implémentation simple des suspensions satisfait ces règles.

Les *invariants* et *l'état fantôme monotone* d'Iris le permettent  
**au moins pour partie.** 🐼

– *cf. séminaire de Jacques-Henri Jourdan*

Okasaki (1999) imagine l'analyse amortie à base de débits.

Danielsson (2008) formalise cette analyse en termes de typage.

Pilkiewicz et Pottier (2011) reconstruisent les débits à partir des crédits.

Mével et al. (2019) reproduisent cette construction dans le cadre d'Iris.

Madhavan et al. (2017) développent un système automatisé pour vérifier des programmes avec mémoisation.

- ① Un temps global en logique de Hoare
- ② Un temps local en logique de séparation : les crédits
- ③ Un temps pour la paresse : les débits
- ④ Un souvenir du temps passé : les reçus
- ⑤ Conclusion

Les crédits-temps sont *consommés* au cours du temps :

$$\{ \$1 \} \text{ tick}() \{ True \}$$

Ils garantissent une *borne supérieure* sur la durée de l'exécution :

Si  $\{ \$n \} p \{ True \}$  alors  $p$  n'exécutera pas plus de  $n$  ticks.

Les crédits-temps nous parlent du temps restant, du temps *futur*.

Et si nous renversions tout cela ?

Imaginons des *reçus* qui sont *produits* au cours du temps :

$$\{ True \} \text{ tick}() \{ \text{⌚} 1 \}$$

Ils garantissent une *borne inférieure* sur la durée de l'exécution :

Si  $\{ True \} p \{ \text{⌚} n \}$  alors  $p$  exécute au moins  $n$  ticks.

Les reçus-temps nous parlent du temps *passé*.

Pourquoi s'efforcer de prouver qu'un programme est *lent* ?

Pourquoi s'efforcer de prouver qu'un programme est *lent* ?

Si à un certain point ce programme risque d'échouer,  
alors en démontrant qu'il est très lent,  
on montre que *l'échec est très lointain*.

Pourquoi s'efforcer de prouver qu'un programme est *lent* ?

Si à un certain point ce programme risque d'échouer,  
alors en démontrant qu'il est très lent,  
on montre que *l'échec est très lointain*.

On peut même montrer que l'échec est *impossible*

Pourquoi s'efforcer de prouver qu'un programme est *lent* ?

Si à un certain point ce programme risque d'échouer,  
alors en démontrant qu'il est très lent,  
on montre que *l'échec est très lointain*.

On peut même montrer que l'échec est *impossible*  
si l'on admet que *le temps est fini* :

$$\exists N \Rightarrow \text{False}$$

On fixe une fois pour toutes un entier  $N$  très grand, par exemple  $2^{64}$ .

Par contraposition, cet axiome donne

$$\exists n \Rightarrow n < N$$

Si on peut prouver que l'on a déjà effectué  $n$  pas de calcul,  
alors on peut affirmer que  $n$  est *petit*.

À quoi cela peut-il servir ?

Par contraposition, cet axiome donne

$$\nexists n \Rightarrow n < N$$

Si on peut prouver que l'on a déjà effectué  $n$  pas de calcul, alors on peut affirmer que  $n$  est *petit*.

À quoi cela peut-il servir ?

À démontrer l'absence de débordements entiers, par exemple.  
– *un argument de complexité aide à la preuve de sûreté!*

# Spécification de l'addition des entiers machine

En arithmétique 64 bits signée, l'addition peut être spécifiée ainsi :

$$\begin{aligned} &\{-2^{63} \leq n_1 + n_2 < 2^{63}\} \\ &\quad \text{add}(n_1, n_2) \\ &\quad \{\lambda n. n = n_1 + n_2\} \end{aligned}$$

On a *l'obligation de prouver* que le résultat  $n_1 + n_2$  est représentable.

# Une situation où le débordement est impossible

Imaginons que les entiers machine  $n_1$  et  $n_2$   
que nous souhaitons additionner  
sont les tailles de deux structures que nous avons construites.

Il se pourrait dans ce cas que nous disposions de  $\llbracket n_1$  et  $\llbracket n_2$ .

Nous pouvons en tirer  $\llbracket(n_1 + n_2)$  puis déduire  $0 \leq n_1 + n_2 < N$ .

Si  $2^{63} \leq N$ , cela garantit l'absence de débordement.

# Une situation où le débordement est impossible

Pour résumer, si on choisit  $N$  assez grand :

$$2^{63} \leq N$$

alors on peut *démontrer* pour l'addition machine cette spécification :

$$\begin{aligned} & \{ \text{⌈} n_1 \star \text{⌋} n_2 \} \\ & \quad \text{add}(n_1, n_2) \\ & \{ \lambda n. n = n_1 + n_2 \star \text{⌈} n \} \end{aligned}$$

En bref, l'addition de *deux horloges disjointes* ne peut pas déborder.

On peut ainsi démontrer que certains entiers, par exemple une *taille* ou une *profondeur*, sont toujours représentables par un mot machine.

Clochard et al. (2015) proposent un langage doté de certains types de compteurs entiers restreints et montrent que ces compteurs ne peuvent pas déborder.

Mével et al. (2019) réexpliquent cette idée en termes de reçus-temps en démontrent la correction dans Iris.

- ① Un temps global en logique de Hoare
- ② Un temps local en logique de séparation : les crédits
- ③ Un temps pour la paresse : les débits
- ④ Un souvenir du temps passé : les reçus
- ⑤ Conclusion

De mauvaises et de bonnes nouvelles :

- *la vérification de programmes est difficile*  
et demande des concepts et des outils pointus,
- mais *raisonner à propos du temps*  
n'est pas beaucoup plus difficile !

Intégrer la complexité dans les spécifications : *viable* à grande échelle ?

Exploiter des analyses *automatiques* (Carbonneaux et al., 2017) tout en conservant la possibilité d'argumenter plus finement là où nécessaire.

Vérifier la complexité d'algorithmes *randomisés* (Eberl et al., 2020).

Vérifier la vivacité de programmes *concurrents* (Hoffmann et al., 2013).

Contrôler le *carburant* dans la blockchain (Das et Qadeer, 2020).

Appendice :  
Une application potentielle

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.

**Example 13** (Many function coarsest partition problem). The many function coarsest partition problem inputs a finite set  $s$ , an initial partition  $P$  of  $s$ , and a family of total functions  $h_i$  on  $s$ ,  $i = 1, \dots, k$ . It outputs the coarsest (i.e., greatest) refinement  $Q$  of  $P$  such that  $\forall b \in Q \forall i = 1, \dots, k \exists d \in Q | h_i[b] \subseteq d$ . It is straightforward to reformulate this problem as computing the greatest common fixed point (that is a refinement of  $P$ ) of the family of functions

$$g_i(Q) = \{b \cap h_i^{-1}[q] : b \in Q, q \in Q | b \cap h_i^{-1}[q] \neq \{\}\},$$

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.

... (subset partition problem). The many function  
... of  $s$  and a

This leads to the algorithm just below,

```
Q = {s}
(while  $\exists q \in (Q - f(Q)) \exists b \in f(Q) | b \subset q$  and  $\#b \leq \#q/2$ )
  Q =  $(Q - \{q\}) \cup \{q - b, b\}$ 
end
```

(33)

$g_i(Q) = \{b \cap h_i^{-1}[q] : b \in Q, q \in Q\}$

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.

This leads to the algorithm for the *subset partition problem*. The many function and a

```
Q = {s}
(while  $\exists q \in (Q - P) \exists b \in P | b \subset q$  and  $\#b \leq \#q/2$ )
  (for  $t = 1, \dots, k$ )
     $P := split_t(P, b)$ 
  end
   $Q = (Q - \{q\}) \cup \{q - b, b\}$ 
end
```

(33)

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.
- *Hopcroft (1971)* obtient une borne  $O(kn \log n)$ .

```

WHILE SPLIT(BSPLIT)≠0 DO
  BEGIN
    STATE:=DATA(SPLIT(BSPLIT));
    REMOVEFROM(SPLIT(BSPLIT));
    STATENEXT(STATELAST(N+STATE)):=STATENEXT(N+STATE);
    IF STATENEXT(N+STATE)≠0 THEN
      STATELAST(STATENEXT(N+STATE)):=STATELAST(N+STATE);
    INSERTSTATE(FREEBLOCK,STATE);
    BLOCK(STATE):=FREEBLOCK;
  IF PREVNZERO(STATE)≠0 THEN
    BEGIN
      ZEROEXT(ZEROLAST(N+STATE)):=ZERONEXT(N+STATE);
      IF ZEROEXT(N+STATE)≠0 THEN
        ZEROEXT(ZERONEXT(N+STATE)):=ZEROLAST(N+STATE);
      INSERTZERO(FREEBLOCK,STATE);
      NUMZERO(BSPLIT):=NUMZERO(BSPLIT)-NZERCTR(STATE);
      NUMZERO(FREEBLOCK):=NUMZERO(FREEBLOCK)+NZERCTR(STATE);
    END;
  END;

```

lication potentielle

ral dans le domaine

é  $O(kn^2)$ .  
irement.

- Hopcroft (1971) obtient une borne  $O(kn \log n)$ .

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.
- *Hopcroft (1971)* obtient une borne  $O(kn \log n)$ .
- *Knuutila (2001)* affirme que certaines variantes violent cette borne.

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

... (1956) a une complexité  $O(kn^2)$ .  
proofs and a well-founded computational analysis. Our analysis reveals that if the size of the input alphabet  $m$  is not fixed, then Hopcroft's original algorithm does not run in time  $O(mn \log n)$  as is commonly believed in the literature. The  $O(mn \log n)$  holds, however, for the variation presented later by D. Gries and for a new variant given in this article. We also propose a new

- Knuutila (2001) affirme que certaines variantes violent cette borne.

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.
- *Hopcroft (1971)* obtient une borne  $O(kn \log n)$ .
- *Knuutila (2001)* affirme que certaines variantes violent cette borne.
- *Valmari (2012)* propose un algorithme de coût  $O(n + m \log m)$ .

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

(1956) complexité  $O(kn^2)$   
Minimization of deterministic finite automata has traditionally required complicated programs and correctness proofs, and taken  $O(nk \log n)$  time, where  $n$  is the number of states and  $k$  the size of the alphabet. Here a short, memory-efficient program is presented that runs in  $O(n + m \log m)$ , or even in  $O(n + m \log n)$ , time, where  $m$  is the number of transitions. The program is complete with input, output, and the removal of irrelevant parts of the automaton. Its invariant-style correctness proof is relatively short.

- Valmari (2012) propose un algorithme de coût  $O(n + m \log n)$

La *minimisation d'*  
du *model-checking*.

Minimization of dete  
programs and correctn  
states and  $k$  the size of  
that runs in  $O(n + m)$   
transitions. The program  
parts of the automaton.

- Valmari (2012)

```
void split(){
  while( w ){
    int s = W[--w], j = F[s]+M[s];
    if( j == P[s] ){M[s] = 0; continue;}
    if( M[s] <= P[s]-j ){
      F[z] = F[s]; P[z] = F[s] = j; }
    else{
      P[z] = P[s]; F[z] = P[s] = j; }
    for( int i = F[z]; i < P[z]; ++i ){
      S[E[i]] = z; }
    M[s] = M[z++] = 0;
  }
}
```

La *minimisation d'automates finis* joue un rôle central dans le domaine du *model-checking*.

- L'algorithme de Moore (1956) a une complexité  $O(kn^2)$ .
  - *Cai et Paige (1989)* le décrivent très clairement.
- *Hopcroft (1971)* obtient une borne  $O(kn \log n)$ .
- *Knuutila (2001)* affirme que certaines variantes violent cette borne.
- *Valmari (2012)* propose un algorithme de coût  $O(n + m \log m)$ .

Une littérature et un algorithme difficiles à maîtriser !

*Vérifier* la complexité d'*une implémentation réelle* serait utile.

## Appendice : Synthèse semi-interactive d'une fonction de coût

Rappelons-nous la fonction *bsearch* de recherche dichotomique d'un élément  $x$  dans un segment  $[i, j)$  du tableau trié  $a$ .

On suppose qu'un *tick()* a été inséré au début de la fonction.

On veut énoncer puis démontrer que

- *bsearch a i j x* exécute  $O(\log n)$  ticks.

Qu'est-ce que cela signifie? Qu'est-ce que  $n$ ?

Rappelons-nous la fonction *bsearch* de recherche dichotomique d'un élément  $x$  dans un segment  $[i, j)$  du tableau trié  $a$ .

On suppose qu'un *tick()* a été inséré au début de la fonction.

On veut énoncer puis démontrer que

- *bsearch a i j x* exécute  $O(\log n)$  *ticks*, où  $n = j - i$ .

*On choisit* les paramètres de l'analyse de complexité, ici la taille du segment de tableau pertinent.

Qu'est-ce que cela signifie ? Qu'est-ce que  $O$  ?

On veut affirmer qu'*il existe une fonction de coût  $f$*  telle que

- *$bsearch$  a  $i j x$  exécute au plus  $f(n)$  ticks où  $n = j - i$ ;*
- *$f(n) = O(\log n)$ .*

On veut affirmer qu'*il existe une fonction de coût  $f$*  telle que

- *$bsearch$  a  $i j x$  exécute au plus  $f(n)$  ticks où  $n = j - i$ ;*
- *$f(n)$  est  $O(\log n)$ .*

On veut affirmer qu'*il existe une fonction de coût*  $f$  telle que

- *bsearch* a  $i j x$  exécute au plus  $f(n)$  ticks où  $n = j - i$  ;
- $f$  est  $O(\log)$ .

On veut affirmer qu'*il existe une fonction de coût*  $f$  telle que

- *bsearch* a  $i$   $j$   $x$  exécute au plus  $f(n)$  ticks où  $n = j - i$  ;
- $f \preceq \log$ .

La fonction  $f$  est *dominée* asymptotiquement par la fonction  $\log$ .

On veut affirmer qu'*il existe une fonction de coût*  $f$  telle que

- *bsearch* a  $i$   $j$   $x$  exécute au plus  $f(n)$  ticks où  $n = j - i$ ;
- $f \preceq \log$ .

D'où deux questions :

- comment *synthétiser*  $f$  ou une description de  $f$  à partir du code ?
- qu'est-ce que la domination asymptotique ?

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

$f(j-i) \geq 1 + ??$

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if j <= i then ?? else
  ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if j <= i then ?? else
  ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then ?? else
  ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
  ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
      if x = Array.get a k then k
      else if x < Array.get a k
        then bsearch a x i k
        else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max ?? ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 ??
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 (1 + ??)
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
  let k = i + (j - i) / 2 in
  if x = Array.get a k then k
  else if x < Array.get a k
    then bsearch a x i k
    else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
  0 + 1 + max 0 (1 + max ?? ??)
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 (
      1 + max (f (k-i)) ??
    )
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 (
      1 + max (f (k-i)) ??
    )
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 (
      1 + max (f ((j-i)/2)) ??
    )
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f (j-i) >= 1 + (
  if (j-i) <= 0 then 0 else
    0 + 1 + max 0 (
      1 + max (f ((j-i)/2))
              (f ((j-i) - (j-i)/2 - 1))
    )
)
```

# Synthèse semi-interactive d'une inéquation récursive

```
let rec bsearch a x i j = (* tick(); *)
  if j <= i then -1 else
    let k = i + (j - i) / 2 in
    if x = Array.get a k then k
    else if x < Array.get a k
      then bsearch a x i k
      else bsearch a x (k+1) j
```

---

```
f n    >= 1 + (
          if n <= 0 then 0 else
            0 + 1 + max 0 (
              1 + max (f (n/2))
                      (f (n - n/2 - 1))
            )
        )
```

Il reste à *exhiber* une fonction  $f$  telle que

$$f(n) \geq 1 + \begin{cases} 0 & \text{si } n \leq 0 \\ 1 + \max(0, 1 + \max(f(\frac{n}{2}), f(n - \frac{n}{2} - 1))) & \text{sinon} \end{cases}$$

et à *démontrer*  $f \preceq \log$ .

On le fait traditionnellement soit via la *méthode de substitution*,  
soit via le *Master Theorem*.

Pour  $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ , la notation  $f \preceq g$  signifie :

$$\exists k \quad \exists x_0 \quad \forall x \geq x_0 \quad |f(x)| \leq k |g(x)|$$

Pour  $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ , la notation  $f \preceq g$  signifie :

$$\exists k \quad \exists x_0 \quad \forall x \geq x_0 \quad |f(x)| \leq k |g(x)|$$

On peut la généraliser pour  $f, g : A \rightarrow B$  :

$$\exists k \quad \forall x \quad |f(x)| \leq k |g(x)|$$

Pour  $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ , la notation  $f \preceq g$  signifie :

$$\exists k \exists x_0 \forall x \geq x_0 |f(x)| \leq k |g(x)|$$

On peut la généraliser pour  $f, g : A \rightarrow B$  :

$$\exists k \text{ AG}_x |f(x)| \leq k |g(x)|$$

AG doit être un *filtre*, un objet de type  $\mathcal{P}(\mathcal{P}(A))$  tel que

Pour  $f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ , la notation  $f \preceq g$  signifie :

$$\exists k \exists x_0 \forall x \geq x_0 |f(x)| \leq k |g(x)|$$

On peut la généraliser pour  $f, g : A \rightarrow B$  :

$$\exists k \text{ AG}_x |f(x)| \leq k |g(x)|$$

AG doit être un *filtre*, un objet de type  $\mathcal{P}(\mathcal{P}(A))$  tel que

$(\forall x P x \rightarrow Q x) \rightarrow (\text{AG}_x P x) \rightarrow (\text{AG}_x Q x)$	covariance
$(\text{AG}_x \text{ True})$	0-intersection
$(\text{AG}_x P x) \rightarrow (\text{AG}_x Q x) \rightarrow (\text{AG}_x P x \wedge Q x)$	2-intersection
$(\text{AG}_x P x) \rightarrow (\exists x P x)$	non vide

Il existe en général de nombreux filtres *incomparables*.

Eberl (2016) (2019) formalise le Master Theorem et automatise les preuves de domination.

Guéneau (2019) formalise les filtres et la domination en Coq.

- Atkey, R. 2011. *Amortised resource analysis with separation logic*. *Logical Methods in Computer Science* 7, 2 :17.
- Carbonneaux, Q., Hoffmann, J., Reps, T., et Shao, Z. 2017. *Automated resource analysis with Coq proof objects*. In *Computer Aided Verification (CAV)*. Lecture Notes in Computer Science, vol. 10427. Springer, 64–85.
- Charguéraud, A. et Pottier, F. 2017. *Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits*. *Journal of Automated Reasoning*.
- Clochard, M., Filliâtre, J.-C., et Paskevich, A. 2015. *How to avoid proving the absence of integer overflows*. In *Verified Software : Theories, Tools and Experiments*. Lecture Notes in Computer Science, vol. 9593. Springer, 94–109.

- Crary, K. et Weirich, S. 2000. **Resource bound certification**. In *Principles of Programming Languages (POPL)*. 184–198.
- Danielsson, N. A. 2008. **Lightweight semiformal time complexity analysis for purely functional data structures**. In *Principles of Programming Languages (POPL)*.
- Das, A. et Qadeer, S. 2020. **Exact and linear-time gas-cost analysis**. In *Static Analysis Symposium (SAS)*. Lecture Notes in Computer Science, vol. 12389. Springer, 333–356.
- Dockins, R. et Hobor, A. 2012. **Time bounds for general function pointers**. In *Mathematical Foundations of Programming Semantics*. Electronic Notes in Theoretical Computer Science, vol. 286. Elsevier, 139–155.

- Eberl, M., Haslbeck, M. W., et Nipkow, T. 2020. **Verified analysis of random binary tree structures**. *Journal of Automated Reasoning* 64, 5, 879–910.
- Guéneau, A., Jourdan, J.-H., Charguéraud, A., et Pottier, F. 2019. **Formal proof and analysis of an incremental cycle detection algorithm**. In *Interactive Theorem Proving (ITP)*. Leibniz International Proceedings in Informatics, vol. 141. 18 :1–18 :20.
- Haslbeck, M. P. L. et Lammich, P. 2021. **For a few dollars more - verified fine-grained algorithm analysis down to LLVM**. In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 12648. Springer, 292–319.
- Haslbeck, M. P. L. et Nipkow, T. 2018. **Hoare logics for time bounds: A study in meta theory**. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 10805. Springer, 155–171.

- Hoffmann, J., Marmar, M., et Shao, Z. 2013. *Quantitative reasoning for proving lock-freedom*. In *Logic in Computer Science (LICS)*. 124–133.
- Madhavan, R., Kulal, S., et Kuncak, V. 2017. *Contract-based resource verification for higher-order functions with memoization*. In *Principles of Programming Languages (POPL)*. 330–343.
- Mével, G., Jourdan, J.-H., et Pottier, F. 2019. *Time credits and time receipts in Iris*. In *European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 11423. Springer, 1–27.
- Nipkow, T. 2015. *Amortized complexity verified*. In *Interactive Theorem Proving (ITP)*. Lecture Notes in Computer Science, vol. 9236. Springer, 310–324.
- Okasaki, C. 1999. *Purely Functional Data Structures*. Cambridge University Press.

- Pilkiewicz, A. et Pottier, F. 2011. **The essence of monotonic state**. In *Types in Language Design and Implementation (TLDI)*.
- Tarjan, R. E. 1985. **Amortized computational complexity**. *SIAM Journal on Algebraic and Discrete Methods* 6, 2, 306–318.
- Zhan, B. et Haslbeck, M. P. L. 2018. **Verifying asymptotic time complexity of imperative programs in Isabelle**. In *International Joint Conference on Automated Reasoning*.