Peut-on dupliquer un objet ?

Linéarité et contrôle des ressources

Guillaume Munch-Maccagnoni

Inria

19 décembre 2018
Collège de France

Qu’est-ce que la programmation système ?

Origine : besoin de gérer les ressources limitées d’une machine partagées entre plusieurs programmes et utilisateurs.

Mémoire		Allouer dynamiquement
Descripteur de fichier		Partager
Interface de connexion		Garantir l’utilisation correcte

Qu'est-ce que la programmation système ?

	acquérir	→ utiliser →	**libérer**
Mémoire	`malloc()`		`free()`
Fichier	`fopen()`		`fclose()`

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ *gestion de la mémoire*)
- abstractions de bas niveau (continuations…)
- valeur qui demande une action de nettoyage
  - descripteur de fichier
  - connexion
  - verrou
  - valeur d'un autre langage de programmation
- accès unique en écriture dans une valeur
- …n'importe quelle structure de données en contenant (listes de ressources, clôtures contenant des ressources…)

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ ***gestion de la mémoire***)
- abstractions de bas niveau (continuations…)
- valeur qui demande une action de nettoyage
  - descripteur de fichier
  - connexion
  - verrou
  - valeur d'un autre langage de programmation
- accès unique en écriture dans une valeur
- …n'importe quelle structure de données en contenant (listes de ressources, clôtures contenant des ressources…)

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ *gestion de la mémoire*)
- **abstractions de bas niveau** (continuations…)
- valeur qui demande une action de nettoyage
  - descripteur de fichier
  - connexion
  - **verrou**
  - valeur d'un autre langage de programmation
- **accès unique en écriture dans une valeur**
- …n'importe quelle structure de données en contenant (listes de ressources, clôtures contenant des ressources…)

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ *gestion de la mémoire*)
- abstractions de bas niveau (continuations…)
- valeur qui demande une action de nettoyage
  - descripteur de fichier
  - connexion
  - verrou
  - **valeur d'un autre langage de programmation**
- accès unique en écriture dans une valeur
- …n'importe quelle structure de données en contenant (listes de ressources, clôtures contenant des ressources…)

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ *gestion de la mémoire*)
- abstractions de bas niveau (continuations…)
- valeur qui demande une action de nettoyage
  - descripteur de fichier
  - connexion
  - verrou
  - valeur d'un autre langage de programmation
- accès unique en écriture dans une valeur
- …n'importe quelle **structure de données** en contenant (listes de ressources, clôtures contenant des ressources…)

Gérer les ressources manuellement

```
let g res =
  (* ...utilise res... *)
  libérer res

let f () =
  let res = acquérir () in
  (* ... *)
  g res
```

Gérer les ressources manuellement

```
let g res =
  (* ...utilise res... *)
```

```
let f () =
  let res = acquérir () in
  (* ... *)
  g res
```

Fuite

Gérer les ressources manuellement

```
let g res =
  (* ...utilise res... *)
  raise Exit

let f () =
  let res = acquérir () in
  g res;
  libérer res
```

Fuite

Gérer les ressources manuellement

```
let g res =
  (* ...utilise res... *)
  libérer res

let f () =
  let res = acquérir () in
  g res;
  h res (* utilise res *)
```

Utilisation-après-libération (*use-after-free*)

Gérer les ressources manuellement

```
let g (res1, res2) =
  (* ...utilise res1... *)
  libérer res1
  (* ...utilise res2... *)
  libérer res2

let f () =
  let res = acquérir () in
  g (res, res)
```

Utilisation-après-libération (*use-after-free*)

Gérer les ressources manuellement

```
let g res =
  (* ... *)
  libérer res

let f () =
  let res = acquérir () in
  (* ... *)
  fun () -> g res
```

?

Programmation système vs. fonctionnelle

Prog. fonctionnelle	Prog. système
Éviter les effets de bord	Modifier l'état du monde
Prouvé correct (idéalement)	Préserver les invariants du monde en cas d'erreur

Un dialogue de sourds !

Programmation système vs. fonctionnelle

Le ramasse-miette

Comment permettre la copie et l'effacement des valeurs à volonté *efficacement ?*

Le programme s'arrête à intervalles réguliers pour « ramasser les miettes » : repérer la mémoire allouée qui n'est plus accessible par le programme.

Souvent, le langage permet au programme de spécifier une fonction à exécuter à ce moment-là pour une valeur donnée : un *finaliseur*.

Plus rare en programmation système, mais pas d'objection de principe.

Ressource :

*Valeur qui est difficile à **copier** ou à **effacer**.*

- structures de données grandes ou partagées (⇒ ***gestion de la mémoire***)
- abstractions de bas niveau (continuations…)
- **valeur qui demande une action de nettoyage**
  - descripteur de fichier
  - connexion
  - verrou
  - valeur d'un autre langage de programmation
- accès unique en écriture dans une valeur
- …n'importe quelle **structure de données** en contenant (listes de ressources, clôtures contenant des ressources…)

Programmation système vs. fonctionnelle

```
let f () =
  let res = acquérir () in
  match g res with
  | x -> libérer res ; x
  | exception e -> libérer res ; raise e
```

Programmation système vs. fonctionnelle

unwind-protect (Lisp) et l'idiome *with_*

```
let unwind_protect f g =
  match f () with
  | x -> g () ; x
  | exception e -> g () ; raise e

let with_resource f =
  let res = acquérir () in
  unwind_protect
    (fun () -> f res)
    (fun () -> libérer res)
```

Prévisible et fiable, mais limité en expressivité !
(bien parenthésé)

Les destructeurs

Relecture de unwind-protect comme une abstraction de type : associer un *destructeur* à un type pour définir un nouveau type (de ressources). Un destructeur est une fonction de nettoyage appelée de façon prévisible.

Inventé par B. Stroustrup pour pouvoir ajouter des exceptions à un dialecte de C, qui deviendra C++.

Avec C++11, les ressources deviennent des valeurs de première classe : on peut les passer, les retourner, et les insérer dans les structures de données de la bibliothèque standard (qui sont à leur tour des ressources).

Les destructeurs

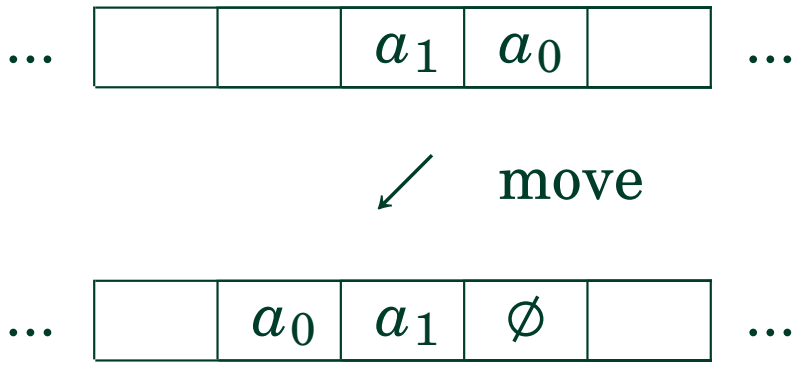
Démos

<1><2><3><4>

Les destructeurs

unique_ptr

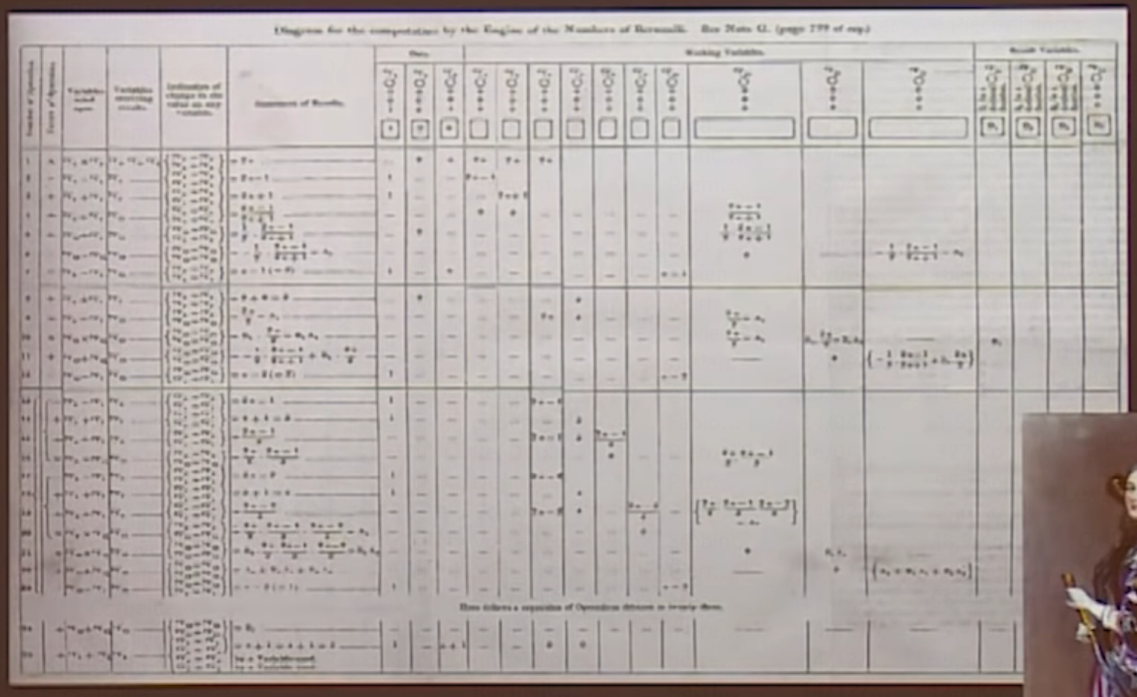
déplacer = copier + mettre à zéro



Retour sur la machine de Babbage…

Le moteur analytique de Babbage (1834)

Jamais réalisé. Ada Lovelace a écrit les premiers programmes.



5

COLLÈGE DE FRANCE 1530

Retour sur la machine de Babbage…

(via http://h14s.p5r.org/2012/11/analytical-programming.html)

Columns on which are inscribed the primitive data.	Number of the operations.	Cards of the operations.		Variable cards.			Statement of results.
		Number of the Operation cards.	Nature of each operation.	Columns acted on by each operation.	Columns that receive the result of each operation.	Indication of change of value on any column.	
$^1V_0 = m$	1	1	×	$^1V_0 \times {}^1V_4 =$	$^1V_6$	$^1V_0 = {}^1V_0$, $^1V_4 = {}^1V_4$	$^1V_6 = m\,n'$
$^1V_1 = n$	2	"	×	$^1V_3 \times {}^1V_1 =$	$^1V_7$	$^1V_3 = {}^1V_3$, $^1V_1 = {}^1V_1$	$^1V_7 = m'\,n$
$^1V_2 = d$	3	"	×	$^1V_2 \times {}^1V_4 =$	$^1V_8$	$^1V_2 = {}^1V_2$, $^1V_4 = {}^0V_4$	$^1V_8 = d\,n'$
$^1V_3 = m'$	4	"	×	$^1V_5 \times {}^1V_1 =$	$^1V_9$	$^1V_5 = {}^1V_5$, $^1V_1 = {}^0V_1$	$^1V_9 = d'\,n$
$^1V_4 = n'$	5	"	×	$^1V_0 \times {}^1V_5 =$	$^1V_{10}$	$^1V_0 = {}^0V_0$, $^1V_5 = {}^0V_5$	$^1V_{10} = d'\,m$
$^1V_5 = d'$	6	"	×	$^1V_2 \times {}^1V_3 =$	$^1V_{11}$	$^1V_2 = {}^0V_2$, $^1V_3 = {}^0V_3$	$^1V_{11} = d\,m'$
	7	2	—	$^1V_6 - {}^1V_7 =$	$^1V_{12}$	$^1V_6 = {}^0V_6$, $^1V_7 = {}^0V_7$	$^1V_{12} = m\,n' - m'\,n$

Retour sur la machine de Babbage…

(via http://h14s.p5r.org/2012/11/analytical-programming.html)

Columns on which are inscribed the primitive data.	Number of the operations.	Cards of the operations.		Variable cards.			Statement of results.
		Number of the Operation cards.	Nature of each operation.	Columns acted on by each operation.	Columns that receive the result of each operation.	Indication of change of value on any column.	
$^1V_0 = n$	1	1	×	$^1V_0 \times {}^1V_4 =$	$^1V_6$	$^1V_0 = {}^1V_0$ $^1V_4 = {}^1V_4$	$^1V_6 = n\,n'$
$^1V_1 = n$	2	"	×	$^1V_2 \times {}^1V_1 =$	$^1V_7$	$^1V_3 = {}^1V_3$ $^1V_1 = {}^1V_1$	$^1V_7 = n'\,n$
$^1V_2 = d$	3	"	×	$^1V_2 \times {}^1V_4 =$	$^1V_8$	$^1V_2 = {}^1V_2$ $^1V_4 = {}^0V_4$	$^1V_8 = d\,n'$
$^1V_3 = n'$	4	"	×	$^1V_5 \times {}^1V_1 =$	$^1V_9$	$^1V_5 = {}^1V_5$ $^1V_1 = {}^0V_1$	$^1V_9 = d'\,n$
$^1V_4 = n'$	5	"	×	$^1V_0 \times {}^1V_5 =$	$^1V_{10}$	$^1V_0 = {}^0V_0$ $^1V_5 = {}^0V_5$	$^1V_{10} = d'\,m$
$^1V_5 = d'$	6	"	×	$^1V_2 \times {}^1V_3 =$	$^1V_{11}$	$^1V_2 = {}^0V_2$ $^1V_3 = {}^0V_3$	$^1V_{11} = d\,m'$
	7	2	—	$^1V_6 - {}^1V_7 =$	$^1V_{12}$	$^1V_6 = {}^0V_6$ $^1V_7 = {}^0V_7$	$^1V_{12} = m\,n' - m'\,n$

Les règles structurelles

Contrôler la copie et l'effacement par le typage ?

Logique linéaire (Girard 1987).

Parti pris : on ne suppose pas que les règles de la logique sont en correspondance avec des règles de typage, ni que ce soit nécessaire pour inspirer des abstractions de programmation.

Les règles structurelles

Calcul des séquents intuitioniste :

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \times B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma, B \vdash C \qquad \Gamma \vdash A}{\Gamma, A \to B \vdash C}$$

Les deux premières règles cachent l'effacement et la copie

Les règles structurelles

Calcul des séquents intuitioniste :

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A \vdash B}{\Gamma, \Delta \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \times B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \times B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma, B \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, A \to B \vdash C}$$

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}$$

Les règles structurelles

Calcul des séquents linéaire intuitioniste :

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A \vdash B}{\Gamma, \Delta \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, B \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, A \multimap B \vdash C}$$

$$\cancel{\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}} \qquad \cancel{\frac{\Gamma \vdash C}{\Gamma, A \vdash C}}$$

Les règles structurelles

Calcul des séquents linéaire intuitioniste :

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Delta, \Gamma' \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, A \multimap B, \Delta, \Gamma' \vdash C}$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash B \multimap A} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, B \multimap A, \Gamma' \vdash C}$$

$$\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C} \qquad \cancel{\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}} \qquad \cancel{\frac{\Gamma, \Gamma' \vdash C}{\Gamma, A, \Gamma' \vdash C}}$$

Les règles structurelles

Calcul de Lambek (1958)

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Delta, \Gamma' \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, A \multimap B, \Delta, \Gamma' \vdash C}$$

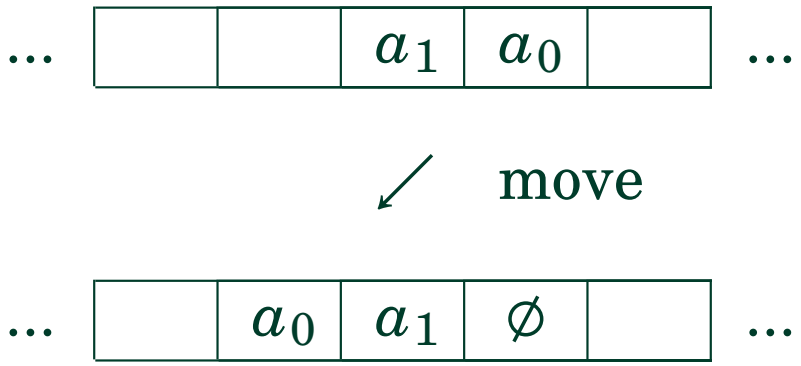
$$\frac{A, \Gamma \vdash B}{\Gamma \vdash B \multimap A} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, B \multimap A, \Gamma' \vdash C}$$

$$\cancel{\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C}} \qquad \cancel{\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}} \qquad \cancel{\frac{\Gamma, \Gamma' \vdash C}{\Gamma, A, \Gamma' \vdash C}}$$

Les règles structurelles

unique_ptr

déplacer = copier + mettre à zéro



Les règles structurelles

H. Baker (1994) :
Le mouvement des ressources de la programmation système peut être implémenté par une permutation de la pile d'appel inspiré de la logique linéaire.

La logique de *unique_ptr*

$$\frac{}{A \vdash A} \qquad \frac{\Delta \vdash A \qquad \Gamma, A, \Gamma' \vdash B}{\Gamma, \Delta, \Gamma' \vdash B}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, A \otimes B, \Delta \vdash C}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, A \multimap B, \Delta, \Gamma' \vdash C}$$

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash B \circ\!\!- A} \qquad \frac{\Gamma, B, \Gamma' \vdash C \qquad \Delta \vdash A}{\Gamma, \Delta, B \circ\!\!- A, \Gamma' \vdash C}$$

$$\frac{\Gamma, A, B, \Gamma' \vdash C}{\Gamma, B, A, \Gamma' \vdash C} \qquad \cancel{\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}} \qquad \frac{\Gamma, \Gamma' \vdash C}{\Gamma, A, \Gamma' \vdash C}$$

La logique de *unique_ptr*

Effets, ressources, et logique modale

Interprétation de la gestion des ressources dans la logique modale S4 (variantes constructive, linéaire).
$\Diamond$: modalité d'effet (ex : état + exceptions)
$\Box$: modalité de ressource (ex : contrôle l'effacement)

$$\Diamond A \to \Box(A \to \Diamond B) \to \Diamond B$$

« Il faut savoir comment arrêter le calcul pour propager une exception »
(En collaboration avec G. Combette)

Linéarité et mise à jour en place

Si on est le seul à avoir accès à une valeur, alors on ne peut pas distinguer entre :

- Oublier cette valeur et en renvoyer une nouvelle
- Modifier la valeur en place

Conséquence : mutation sans effet de bord

Gain en efficacité mais pas en expressivité

Linéarité et mise à jour en place

Exemple : abstraction fonctionnelle des valeurs à trou (Minamide 1998)

Comment représenter des structures de données immuables mais non achevées ? Exemple : implémentation récursive terminale de map.

```
let rec map f = function
  | [] -> []
  | a::l -> let r = f a in r :: (map f l)
```

Idée : représenter la valeur en train d'être construite comme une valeur-fonction (hfun) dont l'application (happ) consiste à boucher le trou. Ne peut être utilisée qu'une seule fois !

```
let rec map_efficace f acc = function
  | [] -> happ(acc,[])
  | a::l -> let r = f a in
            map f (hfun x -> happ(acc,r :: x)) l
```

Emprunts linéaires

Intuition : permission temporaire de lire/écrire dans une valeur mutable ; garantie d'être le seul.

Empêche les invalidations d'itérateur et les courses de données en Rust.

Emprunts linéaires

Démos <1><2>

Emprunts linéaires

Le point de vue de la logique : un nouveau type de types, **copiables en séquence** mais pas en parallèle.

✓ `let y = &mut x in f y; g y`

✗ `let y = &mut x in (y, y)`

Une autre abstraction pour une notion de valeur linéaire (ce que l'on s'interdit de copier est la *permission* d'accéder à la valeur).

Emprunts linéaires

```
val map : ('a ->seq  'b) ->
                        'a list -> 'b list

val parallel_map : ('a ->copy  'b) ->
                        'a list -> 'b list
```

La linéarité dans Curry-Howard

Vers une synthèse de la programmation fonctionnelle et de la programmation système

Des langages de programmation ouverts sur le monde… à l'ère d'Internet !

Une notion de linéarité *des valeurs* ; plusieurs abstractions de typage.

Programmation parallèle et concurrente sans crainte (Rust).

Deux découvertes en logique et en programmation, dont on commence tout juste à en explorer les liens et les conséquences !

La linéarité dans Curry-Howard

Et bien plus...

La logique linéaire a été découverte par J.-Y. Girard via une relecture des modèles du λ-calcul de G. Berry.

La notion de linéarité du calcul permet d'éclaircir les phénomènes d'*ordre d'évaluation*.

Le temps et l'espace sont des ressources ! (Approches théoriques et pratiques à la complexité.)

Modèles de calcul émergents (quantique, probabiliste : C. Tasson le 23 janvier).

La linéarité dans Curry-Howard

Pour aller plus loin...

- Le langage Rust : https://www.rust-lang.org/
- Henry G. Baker. *Linear Logic and Permutation Stacks — The Forth Shall Be First*. SIGARCH Comp. Arch. News, 1994 http://www.pipeline.com/~hbaker1/Use1Var.html *"Use-Once" Variables and Linear Objects — Storage Management, Reflection and Multi-Threading*. SIGPLAN Notices, 1995. http://www.pipeline.com/~hbaker1/Use1Var.html
- Paul-André Melliès. *Categorical semantics of linear logic*, Panoramas et Synthèses vol. 27, pp. 15–215. SMF, 2009. http://www.irif.fr/~mellies/papers/panorama.pdf
- Jean-Yves Girard. *The Blind Spot : Lectures on Logic*. European Mathematical Society, 2011. https://www.ems-ph.org/books/book.php?proj_nr=136