Structures de contrôle, quatrième cours

Programmer ses structures de contrôle : continuations et opérateurs de contrôle

Xavier Leroy

2024-02-15

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

La notion de continuation

Étant donné un point de contrôle dans un programme,
la continuation du point de contrôle est

> la suite de calculs qui restent à effectuer
> une fois que l'exécution a atteint le point de contrôle
> pour finir l'exécution du programme.

Souvent, cette continuation peut être représentée dans le langage de programmation, comme une commande, une fonction, etc.

Exemples de continuations

Dans un langage impératif avec contrôle structuré.

Programme	*Continuation de…*
s_1 ❶ ; s_2	❶ s_2
(if be then ❶ s_1 else ❷ s_2); s_3	❶ $s_1; s_3$ ❷ $s_2; s_3$
while be do ❶ s ❷	❶ s; while be do s ❷ while be do s
for $i = 1$ to 10 do ❶ s	❶ s; while $i < 10$ do $(i = i + 1; s)$

Continuations dans un langage fonctionnel

Dans un langage à base d'expressions (langage fonctionnel p.ex.) on parle plutôt de la continuation d'une sous-expression e dans un programme p :

> la continuation de e dans p est
> la suite de calculs qui restent à effectuer
> une fois qu'on a évalué e en sa valeur v_e
> pour finir l'évaluation et obtenir la valeur v_p de p.

On peut voir la continuation comme une fonction $v_e \mapsto v_p$

Exemples de continuations

Dans un langage d'expressions arithmétiques,
avec évaluation de gauche à droite.

Considérons le programme $p = (1 + 2) \times (3 + 4)$.

La continuation de 1 dans p est $\lambda v.\ (v + 2) \times (3 + 4)$.

La continuation de $1 + 2$ dans p est $\lambda v.\ v \times (3 + 4)$.

La continuation de $3 + 4$ dans p est $\lambda v.\ 3 \times v$

(et non pas $\lambda v.\ (1 + 2) \times v \ldots$)

NB : la continuation dépend de la stratégie d'évaluation !
(ici, de la gauche vers la droite).

Continuations et sauts

Des commandes comme `goto`, `break`, `return`, ou `throw` (levée d'exception) peuvent se voir comme changeant la continuation : au lieu de la continuation du point de contrôle qui suit syntaxiquement, on se branche à

`goto` L	la continuation du point étiqueté L
`break`	la continuation de la boucle englobante ;
`return`	la continuation de l'appel de fonction en cours
`throw`	la continuation de la clause `catch` du `try` le plus proche.

Exemple : la continuation de `break` dans

$$\texttt{while } be \texttt{ do } (\texttt{break}; s_1); s_2$$

est s_2 et non pas $s_1; \texttt{while } \ldots; s_2$

Dans ce cours

Trois modalités d'utilisation des continuations :

- comme outil sémantique
  (notamment pour décrire la sémantique du `goto` non local);
- comme outil de programmation fonctionnelle
  (le ≪style à passage de continuations≫, CPS);
- via l'ajout d'opérateurs de contrôle au langage
  (p.ex le `call/cc` de Scheme).

Les continuations comme outil sémantique

La sémantique dénotationnelle

(C. Strachey, D. Scott, C. Wadsworth, etc, à partir de 1965.)

Associer à chaque élément syntaxique d'un langage
(expression, commande, fonction, …)
un objet mathématique qui décrit précisément sa signification.

Exemple : pour le langage des feuilles de calcul, on définit

$$[\![expr]\!] : \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{environnement}} \to Val$$

$$[\![prog]\!] : \wp(Var \xrightarrow{\text{fin}} Val) \qquad \text{(ensemble des solutions)}$$

par récurrence sur la structure de *expr* et de *prog*.

Sémantique dénotationnelle des feuilles de calcul

Expressions : $[\![expr]\!](Var \xrightarrow{\text{fin}} Val) \to Val$

$$[\![cst]\!]\ \rho = cst$$
$$[\![x]\!]\ \rho = \rho(x)$$
$$[\![f(e_1, \ldots, e_n)]\!]\ \rho = f^*([\![e_1]\!]\ \rho, \ldots, [\![e_n]\!]\ \rho)$$

Programmes : $[\![prog]\!] : \wp(Var \xrightarrow{\text{fin}} Val)$

$$[\![x_1 = e_1, \ldots, x_n = e_n]\!] = \{\rho \mid \rho(x_i) = [\![e_i]\!]\ \rho \text{ pour } i = 1, \ldots, n\}$$

Sémantique dénotationnelle de l'affectation

Quel sens donner à l'affectation $x := x + 1$?

Idée : comme un transformateur d'états mémoire (*stores*).

$$[\![stmt]\!] : \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{état mémoire «avant»}} \to \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{état mémoire «après»}}$$

Quelques cas :

$$[\![x := e]\!]\,\sigma = \sigma[x \leftarrow [\![e]\!]\,\sigma]$$

$$[\![s_1; s_2]\!]\,\sigma = [\![s_2]\!]\,([\![s_1]\!]\,\sigma)$$

$$[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]\,\sigma = \begin{cases} [\![s_1]\!]\,\sigma & \text{si } [\![be]\!]\,\sigma = \texttt{true} \\ [\![s_2]\!]\,\sigma & \text{si } [\![be]\!]\,\sigma = \texttt{false} \end{cases}$$

Sémantique dénotationnelle des boucles

Idée : ajouter une dénotation spéciale $\bot$ pour la divergence.

$$[\![stmt]\!] : \overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{état mémoire «avant»}} \to \left(\overbrace{(Var \xrightarrow{\text{fin}} Val)}^{\text{état mémoire «après»}} + \overbrace{\{\bot\}}^{\text{divergence}} \right)$$

On prend alors :

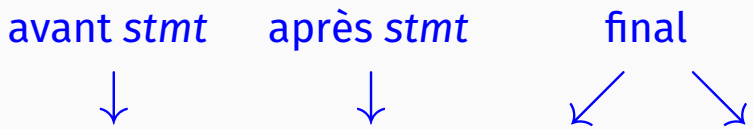
$$[\![\texttt{while } be \texttt{ do } s]\!] = \text{lfp}\,(\lambda d.\lambda\sigma.\texttt{if } [\![be]\!]\, s \texttt{ then } d([\![s]\!]\,\sigma) \texttt{ else } \sigma)$$

où «lfp» est le plus petit point fixe.

Sémantique dénotationnelle des étiquettes et des «goto»

(F. L. Morris, 1970 ; Wadsworth and Strachey, 1970 ; ...)

Idée : la dénotation d'une commande prend en argument explicite la continuation de cette commande. Cela permet de capturer la contination d'une étiquette et de l'associer à l'étiquette dans l'environnement.



$$[\![stmt]\!] : Env \to Store \to (Store \to Res) \to Res$$

$$Store = Var \xrightarrow{\text{fin}} Val$$

$$Res = Store + \{\bot\}$$

$$Env = Label \xrightarrow{\text{fin}} (Store \to Res)$$

Sémantique dénotationnelle à continuations

Pour les commandes qui terminent normalement : la continuation est appliquée à l'état mémoire après exécution de la commande, et produit l'état final du programme.

$$[\![x := e]\!]\ \rho\ \sigma\ k = k\ (\sigma[x \leftarrow [\![e]\!]\ \sigma])$$

$$[\![s_1; s_2]\!]\ \rho\ \sigma\ k = [\![s_1]\!]\ \rho\ \sigma\ (\lambda\sigma'.\ [\![s_2]\!]\ \rho\ \sigma'\ k)$$

$$[\![\texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]\ \rho\ \sigma\ k = \begin{cases} [\![s_1]\!]\ \rho\ \sigma\ k & \text{si } [\![be]\!]\ \sigma = \texttt{true} \\ [\![s_2]\!]\ \rho\ \sigma\ k & \text{si } [\![be]\!]\ \sigma = \texttt{false} \end{cases}$$

Sémantique dénotationnelle des étiquettes et des «goto»

`goto` L ignore la continuation courante et saute à la continuation associée à L dans l'environnement.

$$[\![\texttt{goto } L]\!]\ \rho\ \sigma\ k = \rho(L)\ \sigma$$

Une définition de l'étiquette L associe la continuation correspondante à L dans l'environnement.

$$[\![\texttt{begin } s_1;\ L : s_2\ \texttt{ end}]\!]\ \rho\ \sigma\ k = [\![s_1; s_2]\!]\ \rho'\ \sigma\ k$$
$$\text{avec } \rho' = \rho[L \leftarrow k_2]$$
$$\text{et } k_2 = \lambda\sigma'.\ [\![s_2]\!]\ \rho'\ \sigma'\ k$$

Stratégies de réduction pour un langage fonctionnel

Au cours précédent, nous avons vu la nécessité de définir et de garantir la stratégie de réduction utilisée pour l'exécution d'un langage fonctionnel :

- Appel par valeur : l'argument d'une fonction est réduit en une valeur avant d'être substitué dans le corps de la fonction.
- Appel par nom : l'argument d'une fonction est substitué non évalué dans le corps de la fonction. Il sera évalué à chaque fois que la fonction a besoin de sa valeur.
- Appel par nécessité (≪évaluation paresseuse≫) : comme l'appel par nom, mais avec mémoïsation des évaluations. L'argument sera évalué la première fois où la fonction a besoin de sa valeur.

Sémantique dénotationnelle d'un langage fonctionnel

Naïvement :

$$Val = Num + (Val \to Val) + \{\bot\}$$

$$[\![expr]\!] : (Var \xrightarrow{\text{fin}} Val) \to Val$$

$$[\![x]\!]\ \rho = \rho(x)$$

$$[\![\lambda x.\ e]\!]\ \rho = v \mapsto [\![e]\!]\ (\rho[x \leftarrow v])$$

$$[\![e_1\ e_2]\!]\ \rho = ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho)$$

Problème 1 : *Val* est mal défini en théorie des ensembles.

Problème 2 : on ne voit pas bien quelle stratégie est implémentée par l'application sémantique $([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho)$.

Avec des domaines de Scott

Appel par nom :

$$\mathit{Res} \approx \mathit{Num} + \mathit{Fun} + \{\bot\} + \{\mathit{err}\} \text{ et } \mathit{Fun} = \mathit{Res} \overset{\text{cont}}{\to} \mathit{Res}$$

$$[\![e_1\ e_2]\!]\ \rho = \begin{cases} ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho) & \text{si } [\![e_1]\!]\ \rho \in \mathit{Fun} \\ \bot & \text{si } [\![e_1]\!]\ \rho = \bot \\ \mathit{err} & \text{sinon} \end{cases}$$

Appel par valeur :

$$\mathit{Res} \approx \mathit{Val} + \{\bot\} + \{\mathit{err}\} \text{ et } \mathit{Val} \approx \mathit{Num} + \mathit{Fun} \text{ et } \mathit{Fun} = \mathit{Val} \overset{\text{cont}}{\to} \mathit{Res}$$

$$[\![e_1\ e_2]\!]\ \rho = \begin{cases} ([\![e_1]\!]\ \rho)\ ([\![e_2]\!]\ \rho) & \text{si } [\![e_1]\!]\ \rho \in \mathit{Fun} \text{ et } [\![e_2]\!] \in \mathit{Val} \\ \bot & \text{si } [\![e_1]\!]\ \rho = \bot \text{ ou } [\![e_1]\!]\ \rho \in \mathit{Fun} \text{ et } [\![e_2]\!]\ \rho = \bot \\ \mathit{err} & \text{sinon} \end{cases}$$

La transformation CPS

Spécifier la stratégie de réduction avec des continuations

Pour rendre plus explicite l'ordre des calculs, on pourrait ajouter des continuations (sémantiques) dans la sémantique dénotationnelle.

Mais un langage fonctionnel est suffisamment expressif pour qu'on puisse matérialiser les continuations syntaxiquement, par transformation de programmes :

$$\text{langage fonctionnel} \rightarrow \text{fragment} \ll\text{CPS}\gg \text{du langage}$$

La transformation CPS

La transformée d'une expression e est une fonction $\lambda k \ldots$ qui :

- prend en argument une fonction k (la continuation);
- réduit e en une valeur v (suivant une stratégie donnée);
- et finalement applique k à v.

La fonction transformée est en style à passage de continuations (CPS, *Continuation-Passing Style*).

Transformation CPS pour l'appel par valeur

$$\mathcal{V}(cst) = \lambda k.\ k\ cst$$
$$\mathcal{V}(x) = \lambda k.\ k\ x$$
$$\mathcal{V}(\lambda x.\ e) = \lambda k.\ k\ (\lambda x.\ \mathcal{V}(e))$$
$$\mathcal{V}(e_1\ e_2) = \lambda k.\ \mathcal{V}(e_1)\ (\lambda v_1.\ \mathcal{V}(e_2)\ (\lambda v_2.\ v_1\ v_2\ k))$$

Les variables sont liées à des valeurs, d'où $\mathcal{V}(x) = \lambda k.\ k\ x$.

Séquence pour un appel $e_1\ e_2$:
évaluer e_1 en v_1, puis évaluer e_2 en v_2, puis appliquer v_1 à v_2.

Transformation CPS pour l'appel par nom

$$\mathcal{N}(cst) = \lambda k.\ k\ cst$$
$$\mathcal{N}(x) = \lambda k.\ x\ k$$
$$\mathcal{N}(\lambda x.\ e) = \lambda k.\ k\ (\lambda x.\ \mathcal{N}(e))$$
$$\mathcal{N}(e_1\ e_2) = \lambda k.\ \mathcal{N}(e_1)\ (\lambda v_1.\ v_1\ (\mathcal{N}(e_2))\ k))$$

Les variables sont liées à des calculs suspendus, d'où $\mathcal{N}(x) = \lambda k.\ x\ k$ ou juste $\mathcal{N}(x) = x$.

Séquence pour un appel $e_1\ e_2$:
évaluer e_1 en v_1, puis appliquer v_1 au calcul suspendu $\mathcal{N}(e_2)$.

Réductions administratives

Les transformations CPS produisent des termes plus verbeux que ce que l'on écrirait à la main, p.ex. pour l'application d'une variable à une variable :

$$\mathcal{V}(f\ x) = \lambda k.\ (\lambda k_1.k_1\ f)\ (\lambda v_1.\ (\lambda k_2.k_2\ x)\ (\lambda v_2.\ v_1\ v_2\ k))$$

au lieu de $\lambda k.\ f\ x\ k$ tout simplement.

Ce ≪bruit≫ peut être évité en effectuant des ≪réductions administratives≫ $\overset{adm}{\to}$ après la transformation : des β-réductions qui éliminent les ≪redex administratifs≫ introduits par la traduction. En particulier, on peut faire

$$(\lambda k.\ k\ v)\ (\lambda x.\ a) \overset{adm}{\to} (\lambda x.\ a)\ v \overset{adm}{\to} a[x \leftarrow v]$$

dès que v est une valeur ou une variable.

Exemples de transformations CPS

$$\mathcal{V}(f\,(g\,x))$$
$$= \quad \lambda k.\, g\; x\; (\lambda v.\, f\; v\; k)))$$

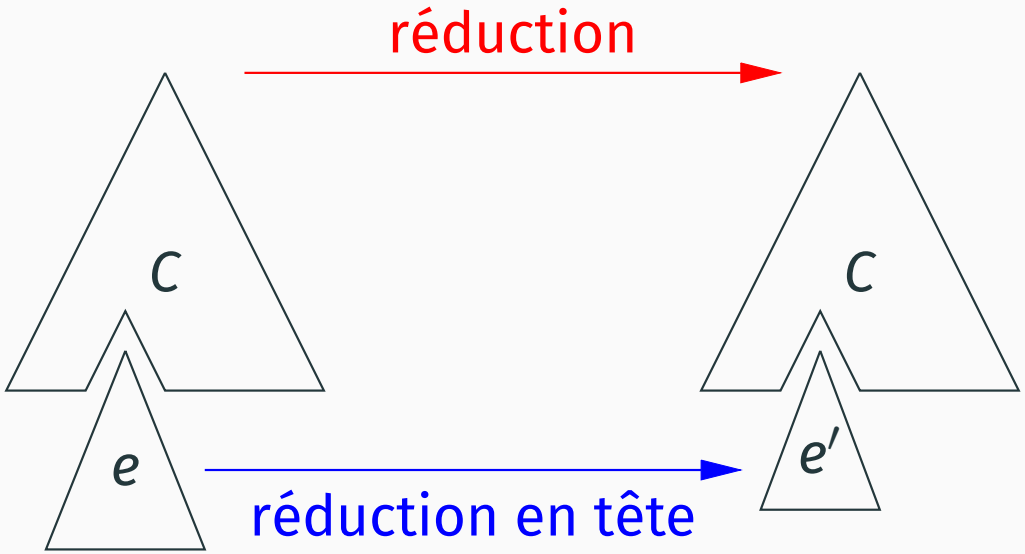
$$\mathcal{N}(f\,(g\,x))$$
$$= \quad \lambda k.\, f\; (\lambda v.\, v\; (\lambda k'.\, g\; (\lambda v'.\, v'\; x\; k'))\; k)$$

$$\mathcal{V}(\texttt{let rec}\ \mathit{fact} = \lambda n.\, \texttt{if}\ \ n = 0\ \texttt{then}\ 1\ \texttt{else}\ \ n * \mathit{fact}(n-1))$$
$$= \quad \texttt{let rec}\ \mathit{fact} = \lambda n.\, \lambda k.$$
$$\texttt{if}\ \ n = 0\ \texttt{then}\ \ k\, 1\ \texttt{else}\ \ \mathit{fact}\,(n-1)\,(\lambda v.\, k\,(n * v)))$$

Spécifier une stratégie de réduction

Par un ensemble de réductions en tête $e \xrightarrow{\varepsilon} e'$
et un ensemble de contextes de réduction C :

$$\frac{e \xrightarrow{\varepsilon} e'}{C[e] \to C[e']}$$



Les stratégies usuelles

Lambda-calcul faible : on peut β-réduire partout sauf sous un λ.

$$(\lambda x.\, e)\ e' \xrightarrow{\varepsilon} e\{x \leftarrow e'\}$$

$$C ::= [\,] \mid C\ e \mid e\ C$$

Appel par nom : pas de réduction des arguments d'applications.

$$(\lambda x.\, e)\ e' \xrightarrow{\varepsilon} e\{x \leftarrow e'\}$$

$$C ::= [\,] \mid C\ e$$

Appel par valeur : réduction gauche-droite des applications ; β-réduction restreinte aux valeurs $v ::= cst \mid \lambda x.\, e$.

$$(\lambda x.\, e)\ v \xrightarrow{\varepsilon} e\{x \leftarrow v\}$$

$$C ::= [\,] \mid C\ e \mid v\ C$$

Correction sémantique de la transformation CPS

(G. Plotkin, *Call-by-name, call-by-value and the lambda-calculus*, TCS 1(2), 1975)

Exécuter un programme e après transformation CPS, c'est appliquer $\mathcal{V}(e)$ ou $\mathcal{N}(e)$ à la continuation initiale $\lambda x.\, x$.

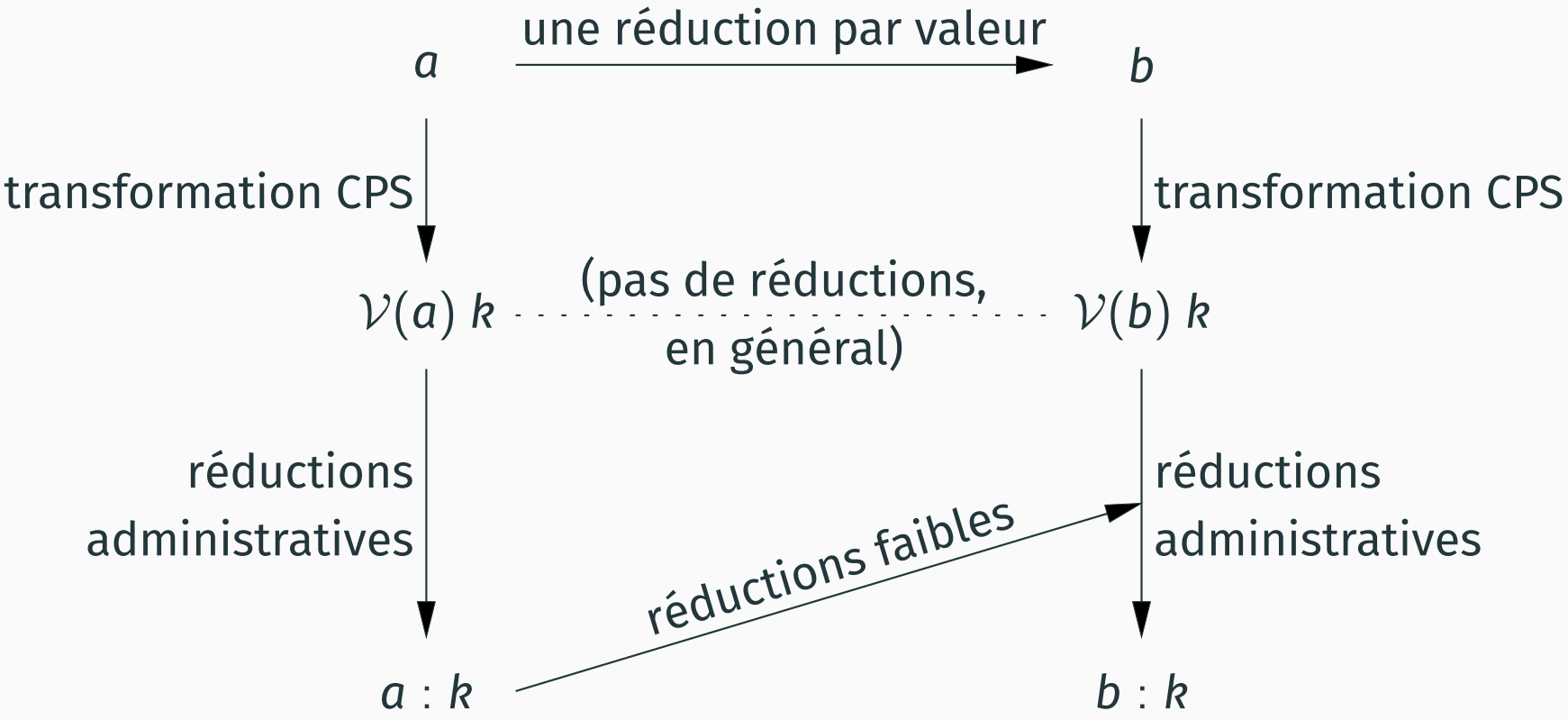
Théorème

Si $e \xrightarrow{*} cst$ *(resp. e diverge) en appel par valeur,*
alors $\mathcal{V}(e)\ (\lambda x.\, x) \xrightarrow{*} cst$ *(resp.* $\mathcal{V}(e)\ (\lambda x.\, x)$ *diverge).*

Si $e \xrightarrow{*} cst$ *(resp. e diverge) en appel par nom,*
alors $\mathcal{N}(e)\ (\lambda x.\, x) \xrightarrow{*} cst$ *(resp.* $\mathcal{N}(e)\ (\lambda x.\, x)$ *diverge) .*

La démonstration de Plotkin

Démonstration subtile utilisant ce diagramme de simulation :



$a : k$, la *colon translation*, est $\mathcal{V}(a)\ k$ où on a réduit des redex administratifs bien choisis.

Les termes CPS

Les termes produits par la transformation CPS ont une forme très particulière, décrite par la grammaire suivante :

$$\begin{aligned} &\text{Atomes :} && a ::= x \mid cst \mid \lambda v.\, b \mid \lambda x.\lambda k.\, b \\ &\text{Corps de fonction :} && b ::= a \mid a_1\ a_2 \mid a_1\ a_2\ a_3 \end{aligned}$$

$\mathcal{V}(e)$ est un atome, et $\mathcal{V}(e)\ (\lambda x.\, x)$ est un corps.

Les applications de fonction (à 1 ou 2 arguments) sont toujours en position terminale.

Réduction des termes CPS

Atomes : $a ::= x \mid cst \mid \lambda v.\ b \mid \lambda x.\lambda k.\ b$

Corps de fonction : $b ::= a \mid a_1\ a_2 \mid a_1\ a_2\ a_3$

Théorème (Indifférence à l'ordre d'évaluation (Plotkin, 1975))

Un programme en forme CPS s'évalue de la même manière en appel par nom, en appel par valeur, et avec toute stratégie faible.

Démonstration.

Partant de $\mathcal{V}(e)$ $(\lambda x.x)$, tous les réduits sont des corps b sans variables libres, c.a.d. v ou $v_1\ v_2$ ou $v_1\ v_2\ v_3$. Les seules réductions possibles dans toute stratégie faible sont $(\lambda x.b)\ v_2 \to b[x \leftarrow v_2]$ et $(\lambda x.\lambda k.b)\ v_2\ v_3 \to (\lambda k.b)[x \leftarrow v_2]\ v_3 \to b[x \leftarrow v_2, k \leftarrow v_3]$. □

Programmer en style à passage de continuations

Programmer en CPS (style à passage de continuation)

Quant on programme dans un langage fonctionnel, il est parfois utile d'appliquer la transformation CPS «à la main» sur certaines parties du programme.

Cela permet de passer explicitement la continuation de l'appel à une fonction de bibliothèque, qui peut s'en servir pour implémenter des structures de contrôle avancées :
itérateurs, coroutines, *threads* coopératifs, …

Itération ≪interne≫ sur un arbre binaire

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

L'itérateur ≪interne≫ usuel en OCaml :

```
let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

Le même, partiellement transformé CPS :

```
let rec tree_iter f t (k: unit -> unit) =
  match t with
  | Leaf -> k ()
  | Node(l, x, r) ->
      tree_iter f l (fun () -> f x; tree_iter f r k)
```

Avantage (?) : il s'exécute en espace de pile constant.

Passage à un itérateur ≪externe≫

Un type général pour énumérer des valeurs à la demande :

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)
```

(Voir aussi : le type `Seq.t` dans la bibliothèque standard d'OCaml.)

Application : itération ≪externe≫ sur un arbre binaire.

```
let rec tree_iter (t: 'a tree) (k: unit -> 'a enum) =
  match t with
  | Leaf -> k ()
  | Node(l, x, r) ->
      tree_iter l (fun () -> More(x, tree_iter r k))

let tree_iterator (t: 'a tree) : 'a enum =
  tree_iter t (fun () -> Done)
```

Utiliser des itérateurs externes

Le *same fringe problem* mentionné au 2e cours.

```
let same_enums (e1: 'a enum) (e2: 'a enum) : bool =
  match e1, e2 with
  | Done, Done -> true
  | More(x1, k1), More(x2, k2) ->
      x1 = x2 && same_enums (k1 ()) (k2 ())
  | _, _ -> false

let same_fringe (t1: 'a tree) (t2: 'a tree) : bool =
  same_enums (tree_iterator t1) (tree_iterator t2)
```

Un générateur avec état dans le style de Python

En ajoutant un état mutable local, on peut transformer cet itérateur en générateur «à la Python», qui renvoie la valeur suivante de l'énumération à chaque fois qu'on l'appelle.

```
exception StopIteration

let tree_generator (t: 'a tree) : unit -> 'a =
  let current = ref (fun () -> tree_iterator t) in
  fun () ->
    match !current () with
    | Done -> raise StopIteration
    | More(x, k) -> current := k; x
```

Une bibliothèque de *threads* coopératifs

L'interface naturelle en ≪style direct≫ :

`spawn : (unit -> unit) -> unit`

Démarre un nouveau *thread*.

`yield : unit -> unit`

Suspend le *thread* courant;
passe la main à un autre *thread*.

`terminate : unit -> unit`

Arrête définitivement le *thread* courant.

Une bibliothèque de *threads* coopératifs

L'interface en CPS (avec une continuation explicite) :

`spawn : (unit -> unit) -> unit`

Démarre un nouveau *thread*.

`yield : (unit -> unit) -> unit`

Suspend le *thread* courant ;
passe la main à un autre *thread*.

`terminate : unit -> unit`

Arrête définitivement le *thread* courant.

Implémentation de la bibliothèque

Une file d'attente de *threads* prêts à s'exécuter.

```
let ready : (unit -> unit) Queue.t = Queue.create ()

let terminate () =
  match Queue.take_opt ready with
  | None -> ()
  | Some k -> k ()

let yield (k: unit -> unit) =
  Queue.add k ready; terminate()

let spawn (f: unit -> unit) =
  Queue.add f ready
```

Un exemple d'utilisation

Afficher les nombres de 1 à `count`, en passant la main :

```
let process name count =
  let rec proc n =
    if n > count then terminate () else begin
      printf "%s%d " name n;
      yield (fun () -> proc (n + 1))
    end
  in proc 1
```

Exemple d'utilisation :

```
let () =
  spawn (fun () -> process "a" 5);
  spawn (fun () -> process "b" 3);
  process "c" 6
```

(Affiche `c1 a1 b1 c2 a2 b2 c3 a3 b3 c4 a4 c5 a5 c6`.)

Continuation de succès, continuation d'échec

Une alternative aux exceptions est d'utiliser un CPS avec deux continuations, l'une appelée si le calcul réussit, l'autre s'il échoue.

Exemple : reconnaissance d'expressions régulières.

```
type 'a regexp =
    char list -> (char list -> 'a) -> (unit -> 'a) -> 'a
```

Le ≪contrat≫ pour une expression régulière R :
$R\ \ell\ k_1\ k_0$ appelle $k_1\ l_2$ si $l = l_1.l_2$ et R a reconnu l_1 ;
$R\ \ell\ k_1\ k_0$ appelle $k_0\ ()$ si R n'a pas reconnu un préfixe de l.

```
let string_match (r: bool regexp) (l: char list) : bool =
  r l (fun l -> l = []) (fun _ -> false)
```

Définition des expressions régulières usuelles

```
let epsilon = fun l k1 k0 -> k1 l

let char (c: char) = fun l k1 k0 ->
    match l with c' :: l' when c' = c -> k1 l' | _ -> k0 ()

let alt (r1: 'a regexp) (r2: 'a regexp) = fun l k1 k0 ->
    r1 l k1 (fun () -> r2 l k1 k0)

let seq (r1: 'a regexp) (r2: 'a regexp) = fun l k1 k0 ->
    r1 l (fun l' -> r2 l' k1 k0) k0

let rec star (r: 'a regexp) = fun l k1 k0 ->
    alt (plus r) epsilon l k1 k0
and plus (r: 'a regexp) = fun l k1 k0 ->
    seq r (star r) l k1 k0
```

Retour en arrière

L'implémentation précédente des expressions régulières n'est pas correcte car les combinateurs sont ≪gloutons≫ et ne peuvent pas revenir en arrière en cas d'échec plus tard. Exemple :

```
string_match (seq (star (char 'a')) (char 'a'))
              ['a'; 'a'; 'a']
```

échoue car `star (char 'a')` a consommé les 3 caractères `a`.

Une meilleure implémentation utilise une seule continuation *k* qui renvoie `true` en cas de succès et `false` pour demander un retour en arrière.

```
type regexp = char list -> (char list -> bool) -> bool
```

Définition des expressions régulières usuelles

```
let epsilon : regexp = fun l k -> k l

let char (c: char) : regexp = fun l k ->
  match l with c' :: l' when c' = c -> k l' | _ -> false

let seq (r1: regexp) (r2: regexp) = fun l k ->
  r1 l (fun l' -> p2 l' k)

let alt (r1: regexp) (r2: regexp) = fun l k ->
  r1 l k || r2 l k

let rec star (r: regexp) : regexp = fun l k ->
  alt (seq r (star r)) epsilon l k

and plus (r: regexp) : regexp = fun l k ->
  seq r (star r) l k
```

Générateurs «internes» et comptage

Un «générateur interne» = une fonction qui produit plusieurs possibilités, les passe à une continuation *k*, et combine les résultats de *k*.

```
let bool k = k false + k true

let rec int lo hi k =
  if lo <= hi then k lo + int (lo + 1) hi k else 0

let rec avltree h k =
  if h < 0 then 0 else if h = 0 then k Leaf else
    avltree2 (h-1) (h-1) k
  + avltree2 (h-2) (h-1) k
  + avltree2 (h-1) (h-2) k
and avltree2 hl hr k =
  avltree hl (fun l -> avltree hr (fun r -> k (Node(l, 0, r))))
```

Générateurs «internes» et comptage

La continuation *k* agit comme une mesure : elle dit combien chaque possibilité contribue à la somme finale.

Ex : compter les arbres AVL de hauteur 4.

```
let n = avltree 4 (fun _ -> 1)
(* 315 *)
```

Ex : compter les jets de dés ≥ 16.

```
let _3d6 k =
  int 1 6 (fun d1 ->
  int 1 6 (fun d2 ->
  int 1 6 (fun d3 -> k (d1,d2,d3))))
let n = _3d6 (fun (d1,d2,d3) ->
                 if d1+d2+d3 >= 16 then 1 else 0)
(* 10 *)
```

Opérateurs de contrôle

Les opérateurs de contrôle

Des constructions de certains langages fonctionnels
qui permettent à une expression de réifier sa continuation,
de la manipuler comme une valeur de 1re classe,
et de relancer cette continuation plus tard.

Permettent de programmer ses structures de contrôle comme on le fait en CPS, en gardant le programme écrit en style ≪direct≫.

ISWIM, Algol, et l'opérateur J

(P. J. Landin, *The next 700 programming languages*, CACM 9, 1966.)
(P. J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation*, CACM 8, 1965.)

Le langage ISWIM : un précurseur de Scheme et de ML.

- Lambda-calcul étendu en appel par valeur.
- Sémantique opérationnelle via la machine abstraite SECD.
- Liaisons statiques des variables via les fermetures ($\neq$ Lisp).

Une explication d'Algol par traduction vers ISWIM étendu :

- État mutable $\rightarrow$ ajout de références (à la ML).
- $\ll$goto$\gg$ non local $\rightarrow$ ajout de l'opérateur de contrôle J.

L'opérateur de contrôle J

$$f = \lambda x.\ \ldots\ \mathtt{J}\,(\lambda y.\, e')\ v\ \ldots$$

L'évaluation de $\mathtt{J}(\lambda y.\, e')\ v$ calcule la valeur de $e'\{y \leftarrow v\}$ et la renvoie directement à l'appelant de f, ≪sautant par dessus≫ les autres calculs dans le corps de f.

Cas particulier : $\mathtt{J}\ (\lambda x.\, x)\ v$ fait comme `return` v dans un langage classique.

Utilisation pour coder les étiquettes et le `goto` :

$$\mathtt{begin}\ s_1;\ L : s_2\ \ \mathtt{end} \ \rightsquigarrow\ \lambda\_.\, \mathtt{let\ rec}\ L = \mathtt{J}\,(\lambda\_.\, s_2)\ \mathtt{in}\ s_1; L\,()$$

$$\mathtt{goto}\ L \ \rightsquigarrow\ L\,()$$

L'opérateur `callcc` (*call with current continuation*)

$$\texttt{callcc}\ (\lambda k.\, e)$$

Une construction du langage Scheme qui capture sa propre continuation, en fait une valeur de fonction, et la passe à $\lambda k.\, e$.

Apparaît sous plusieurs noms dans la littérature :

- J. Reynolds, 1972 : `escape`.
- G. Sussmann et G. Steele, 1975 : `catch` et `throw`.
- Le langage Scheme, à partir de 1982 : `call-with-current-continuation`, abrégé `call/cc`.

Exécution de `callcc`

L'expression `callcc`$(\lambda k.\, e)$ s'évalue comme suit :

- La continuation de cette expression est liée à la variable k.
- e est évaluée ; sa valeur est la valeur de `callcc`$(\lambda k.\, e)$.
- Si, pendant l'évaluation de e ou plus tard, on applique k à une valeur v, l'évaluation continue comme si `callcc`$(\lambda k.\, e)$ avait renvoyé la valeur v.

  En d'autres termes, la continuation de l'expression `callcc` est rétablie et relancée avec v comme résultat pour cette expression.

D’un itérateur ≪interne≫ à un itérateur ≪externe≫

On suppose donné un itérateur ≪interne≫ comme par exemple celui sur les arbres binaires :

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

let rec tree_iter (f: 'a -> unit) (t: 'a tree) =
  match t with
  | Leaf -> ()
  | Node(l, x, r) -> tree_iter f l; f x; tree_iter f r
```

D'un itérateur «interne» à un itérateur «externe»

Avec un `callcc`, on peut interrompre le parcours dès que `tree_iter` a trouvé un élément, et le renvoyer en résultat :

```
let tree_iterator (t: 'a tree) : 'a enum =
  callcc (fun k ->
      tree_iter
          (fun x -> k (Some x))
          t;
      None)
```

L'appel `k (Some x)` interrompt le calcul et fait renvoyer `Some x` comme résultat du `callcc`.

Si l'arbre est vide, la continuation `k` n'est pas appelée et `callcc` renvoie le résultat `None`.

D'un itérateur «interne» à un itérateur «externe»

Avec deux `callcc`, on peut construire un itérateur «externe» (qui énumère les éléments de l'arbre à la demande) au dessus de `tree_iter`.

```
type 'a enum = Done | More of 'a * (unit -> 'a enum)

let tree_iterator (t: 'a tree) : 'a enum =
  callcc (fun k ->
      tree_iter
          (fun x -> callcc (fun k' -> k (More(x, k'))))
          t;
      Done)
```

Si x_1 est l'élément de t le plus à gauche, `tree_iterator` t renvoie `More`(x_1, k_1). Quand k_1 est appelée, l'itération reprend au point où on l'avait laissée, et passe à l'élément suivant de t, ou termine.

Implémenter les exceptions structurées avec `callcc`

En utilisant une pile (impérative) de gestionnaires d'exceptions.

```
let handlers : (exn -> unit) Stack.t = Stack.create()

let raise exn =
  match Stack.pop_opt handlers with
  | Some hdlr -> hdlr exn
  | None -> fatal_error "uncaught exception"

let trywith body hdlr =
    callcc (fun k ->
        Stack.push (fun e -> k (hdlr e)) handlers;
        let res = body () in
        Stack.drop handlers;
        res)
```

Implémenter les exceptions structurées avec `callcc`

La construction

$$\texttt{try } e \texttt{ with } p_1 \to e_n \mid \ldots \mid p_n \to e_n$$

se traduit par

```
trywith
    (fun () -> e)
    (fun exn ->
       match exn with
       | p1 -> e1 | ... | pn -> en
       | _ -> raise exn)
```

Programmer ses structures de contrôle en ≪style direct≫

L'ajout d'opérateurs de contrôle comme `callcc` au langage

- permet de programmer ses propres structures de contrôle sous forme de bibliothèques
  (coroutines, exceptions, *threads* coopératifs, etc),
- tout en gardant le programme en ≪style direct≫
  (pas besoin de le mettre en forme CPS).

Sémantique et implémentation de `callcc`

Sémantique :

- par transformation CPS ;
- directement, avec une sémantique à contextes de réduction.

Implémentation :

- par transformation CPS sur le programme entier ;
- avec plusieurs piles d'appel
  (capturer la cont. = copier la pile ; relancer = changer de pile) ;
- avec une structure de données persistante pour représenter la pile d'appels. ($\rightarrow$ Cours 2022-2023.)

Transformation CPS pour `callcc`

$$\mathcal{V}(\texttt{callcc}\ f) = \lambda k.\ \mathcal{V}(f)\ (\texttt{resume}\ k)\ k$$

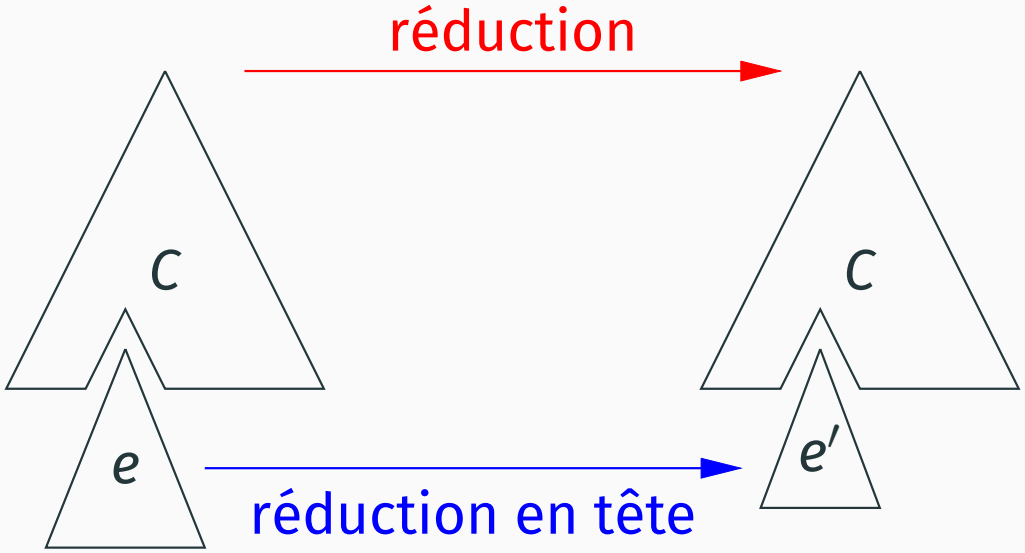
$$\texttt{resume}\ \ k_0 = \lambda v.\ \lambda k.\ k_0\ v$$

La transformation CPS standard utilise les continuations de manière linéaire : chaque k est utilisé exactement une fois.

Pour `callcc` f, on duplique la continuation k :
une fois comme argument de f (dans `resume` k),
et une fois comme continuation de f.

Pour `resume` k_0, on ignore sa continuation k :
l'exécution continue avec k_0.

Continuations et contextes de réduction



Supposons que le programme p se décompose en $p = C[e]$,
avec C un contexte de réduction et e qui se réduit en tête.

Alors la continuation de e dans p n'est autre que $\lambda v.\ C[v]$,
c.à.d. le contexte C réifié comme une fonction. (v non lié dans C)

Règles de réduction pour `callcc`

$$C[\texttt{callcc}(\lambda k.\ e)] \quad \to \quad C[(\lambda k.\ e)\ (\lambda v.\ \texttt{resume}\ C\ v)]$$

$$C[\texttt{resume}\ C_0\ v] \quad \to \quad C_0[v]$$

Ce ne sont pas des réductions sous contexte $\xrightarrow{\varepsilon}$,
mais des réductions $\to$ du programme tout entier.

La règle pour `callcc` duplique le contexte courant C.

La règle pour `resume` le remplace par le contexte capturé C_0.

Continuations délimitées

Les continuations capturées par `callcc` sont non délimitées et abortives : elles s'exécutent jusqu'à la fin du programme et ne reviennent jamais.

Pour certaines applications (retour en arrière, comptage), il est souhaitable d'avoir des continuations délimitées et composables. Par exemple :

$$
\begin{aligned}
& 2 \times \texttt{delim}\,(1 + \texttt{capture}\,(\lambda k.\, k(k\,0))) \\
\xrightarrow{+}\ & 2 \times (\texttt{let}\ \ k = \lambda v.\, 1 + v\ \ \texttt{in}\ \ k(k\,0)) \\
\xrightarrow{+}\ & 2 \times ((1 + (1 + 0))) \xrightarrow{+} 4
\end{aligned}
$$

(Aussi, la capture de continuation peut être moins coûteuse puisque les continuations sont «plus petites».)

Sémantique des continuations délimitées

Des règles de réduction sous contexte de réduction C, mais avec un sous-contexte de réduction D qui ne contient pas `delim` :

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Règles de réduction : (4 variantes !)

$$\begin{aligned} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} (\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v) \\ \texttt{resume}\ D\ v &\xrightarrow{\varepsilon} D[v] \end{aligned}$$

Variante : `-ctrl-`

Sémantique des continuations délimitées

Des règles de réduction sous contexte de réduction C, mais avec un sous-contexte de réduction D qui ne contient pas `delim` :

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Règles de réduction : (4 variantes !)

$$\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} (\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v)$$

$$\texttt{resume}\ D\ v \xrightarrow{\varepsilon} \texttt{delim}(D[v])$$

Variante : `-ctrl-`, `-ctrl+`

Sémantique des continuations délimitées

Des règles de réduction sous contexte de réduction C, mais avec un sous-contexte de réduction D qui ne contient pas `delim` :

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Règles de réduction : (4 variantes !)

$$\begin{aligned} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} \texttt{delim}((\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v)) \\ \texttt{resume}\ D\ v &\xrightarrow{\varepsilon} D[v] \end{aligned}$$

Variante : `-ctrl-`, `-ctrl+`, `+ctrl-`

Sémantique des continuations délimitées

Des règles de réduction sous contexte de réduction C, mais avec un sous-contexte de réduction D qui ne contient pas `delim` :

$$\frac{\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) \xrightarrow{\varepsilon} \ldots}{C[\texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)])] \to C[\ldots]}$$

Règles de réduction : (4 variantes !)

$$\begin{aligned} \texttt{delim}(D[\texttt{capture}\ (\lambda k.\, e)]) &\xrightarrow{\varepsilon} \texttt{delim}((\lambda k.\, e)\ (\lambda v.\, \texttt{resume}\ D\ v)) \\ \texttt{resume}\ D\ v &\xrightarrow{\varepsilon} \texttt{delim}(D[v]) \end{aligned}$$

Variante : `-ctrl-`, `-ctrl+`, `+ctrl-`, `+ctrl+`.

Une ménagerie d'opérateurs de contrôle délimité

(D. Hillerström, réf. en bibliographie.)

Name	Taxonomy	Continuation behaviour	Canonical reference
control/prompt	+**ctrl**−	Composable	Felleisen [81]
shift/reset	+**ctrl**+	Composable	Danvy and Filinski [62]
spawn	−**ctrl**+	Composable	Hieb and Dybvig [116]
splitter	−**ctrl**−	Abortive, composable	Queinnec and Serpette [234]
fcontrol	−**ctrl**−	Composable	Sitaram [250]
cupto	−**ctrl**−	Composable	Gunter et al. [111]
catchcont	−**ctrl**−	Composable	Longley [177]
effect handlers	−**ctrl**+	Composable	Plotkin and Pretnar [228]

Table A.2: Classification of first-class delimited control operators (listed in chronological order).

Point d’étape

Point d'étape

Les continuations sont un puissant concept

- pour comprendre et formaliser la sémantique des sauts non locaux ;
- pour programmer dans les langages fonctionnels en contrôlant l'ordre et l'entrelacement des calculs
  - en style à passage de continuations
  - ou en style direct avec des opérateurs de contrôle.

Voir aussi : les séminaires d'Andrew Kennedy (22/02) et d'Olivier Danvy (29/02).

Voir aussi : les 5e et 6e cours sur les gestionnaires d'effets, une forme moderne et élégante de contrôle délimité.

Bibliographie

Bibliographie

Programmer avec des continuations :

- Daniel P. Friedman and Mitchell Wand, *Essentials of Programming Languages*, MIT Press, 2008. Chapitres 5 et 6.

La ménagerie des opérateurs de contrôle :

- Daniel Hillerström, *Foundations for Programming and Implementing Effect Handlers*, PhD, Edinburgh, 2021. Annexe A, *Continuations*.

Un historique de la notion de continuation :

- John C. Reynolds, *The Discoveries of Continuations*, LISP and Symbolic Computation 6(3–4), 1993.