

Paradigmes et Interprétation

Continuations et Erreurs

Julien Provillard

julien.provillard@univ-cotedazur.fr

CONTINUATIONS

Ordre d'évaluation des expressions

- ☐ Dans les langages que nous avons définis précédemment, nous avons vu que l'ordre d'évaluation des expressions pouvait varier.
- ☐ Dans un langage glouton, les expressions sont évaluées dès qu'elles sont rencontrées.
- ☐ Dans un langage paresseux, les expressions sont évaluées quand cela est nécessaire.

- ☐ Nous allons voir comment prendre le contrôle de l'ordre d'évaluation des expressions.

Évaluation et mise en attente

```
(interp (addE (numE 1) (numE 2)) mt-env)
```

```
→ (num+ (interp (numE 1) mt-env)
    (interp (numE 2) mt-env))
```

```
→ (interp (numE 1) mt-env)
```

```
→ (numV 1)
```

```
→ (interp (numE 2) mt-env)
```

```
→ (numV 2)
```

```
→ (num+ (numV 1) (numV 2))
```

En attente

```
(num+ • (interp (numE 2) mt-env))
```

En attente

```
(num+ • (interp (numE 2) mt-env))
```

En attente

```
(num+ (numV 1) •)
```

En attente

```
(num+ (numV 1) •)
```

Continuation

- ❑ La liste des expressions mises en attente est une **continuation**.

En attente

(+ • 2)

- ❑ Une continuation peut contenir n'importe quelle expression du langage.

En attente

(+ 1 (* • 3))

- ❑ Dans le cas d'une expression composite, on préfère la décomposer.

En attente

(* • 3)

(+ 1 •)

- ❑ L'implémentation classique des continuations est une **pile**.

Représentation des continuations

□ Comment représenter ?

En attente

(+ 1 •)

On mémorise la valeur que l'on devra ajouter.

```
(define-type Cont  
  [doAddK (val : Value)]  
  ... )
```

```
(doAddK (numV 1))
```

Représentation des continuations

❑ Et si on est en train d'évaluer la première expression ?

En attente

(+ • (f 2))

On mémorise l'expression que l'on devra évaluer et son environnement d'évaluation.

```
(define-type Cont
  [addSecondK (r : Exp) (env : Env)]
  [doAddK (val : Value)]
  ... )

(addSecondK (appE (idE 'f) (numE 2)) mt-env)
```

Représentation des continuations

❑ Et lorsque il y a plusieurs éléments dans la continuation ?

En attente

(+ • (f 2))
(+ 1 •)

On pourrait utiliser une liste de continuation.

```
(define-type Cont
  [addSecondK (r : Exp) (env : Env)]
  [doAddK (val : Value)]
  ... )
```


Représentation des continuations

❑ Et lorsque il y a plusieurs éléments dans la continuation ?

En attente

```
(+ • (f 2))
(+ 1 •)
```

On va préférer que chaque continuation connaisse son futur immédiat. Autrement dit le chaînage est codé dans les continuations.

```
(define-type Cont
  [addSecondK (r : Exp) (env : Env) (k : Cont)]
  [doAddK (val : Value) (k : Cont)]
  ... )
```

Représentation des continuations

❑ Et lorsque il y a plusieurs éléments dans la continuation ?

En attente

```
(+ • (f 2))
(+ 1 •)
```

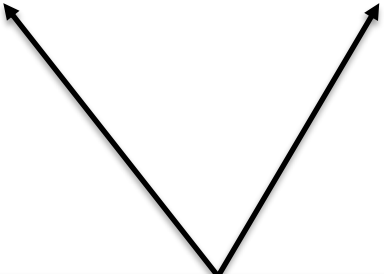
Il faut donc représenter le fait
qu'il n'y ait plus de continuation.

```
(define-type Cont
  [doneK]
  [addSecondK (r : Exp) (env : Env) (k : Cont)]
  [doAddK (v : Value) (k : Cont)]
  ... )
```

```
(addSecondK (appE (idE 'f) (numE 2)) mt-env
  (doAddK (numV 1) (doneK)))
```

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(plusE l r) (num+ (interp l env) (interp r env))]
    ... ))
```



On va dissocier les appels à `interp` pour utiliser les continuations.

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(plusE l r) (interp l env)
                  (num+ • (interp r env))]
    ... ))
```

On va représenter le second appel
à `interp` par une continuation.

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(plusE l r) (interp l env)
                  (addSecondK r env)]
    ... ))
```

Maintenant, il faut connaître le futur d'une évaluation, c'est-à-dire sa continuation.

Et ne pas oublier de passer ce futur à toute continuation que l'on ajouterait.

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(plusE l r) (interp l env)
                  (addSecondK r env k)]
    ... ))
```

Il ne reste plus qu'à passer sa propre continuation à l'appel récursif de `interp`.

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(plusE l r) (interp l env (addSecondK r env k))]
    ... ))
```

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(numE n) (numV n)]
    ... ))
```

Il faut renvoyer la valeur calculée à sa continuation pour continuer le calcul.

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    ... ))
```

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(doneK) val]
    ... ))
```

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    ... ))
```

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(addSecondK r env next-k) (interp r env (doAddK val next-k))]
    ... ))
```

Interpréter avec des continuations

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    ... ))
```

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(doAddK v-l next-k) (continue next-k (num+ v-l val))]
    ... ))
```

Exemples

```
(define (interp e env k)
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    [(plusE l r)
     (interp l env (addSecondK r env k))]
    ... ))
```

```
(interp (numE 1) mt-env (doneK))
```

➡ (continue (doneK) (numV 1))

➡ (numV 1)

```
(define (continue k val)
  (type-case Cont k
    ...
    [(doneK) val]
    [(addSecondK r env next-k)
     (interp r env (doAddK val next-k))]
    [(doAddK v-l next-k)
     (continue next-k (num+ v-l val))]
    ... ))
```

Exemples

```
(define (interp e env k)
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    [(plusE l r)
     (interp l env (addSecondK r env k))]
    ... ))
```

```
(define (continue k val)
  (type-case Cont k
    ...
    [(doneK) val]
    [(addSecondK r env next-k)
     (interp r env (doAddK val next-k))]
    [(doAddK v-l next-k)
     (continue next-k (num+ v-l val))]
    ... ))
```

```
(interp (addE (numE 1) (numE 2)) mt-env
  (doneK))
```

➡

```
(interp (numE 1) mt-env
  (addSecondK (numE 2) mt-env
    (doneK)))
```

➡

```
(continue (addSecondK (numE 2) mt-env
  (doneK))
  (numV 1))
```

➡

```
(interp (numE 2) mt-env
  (doAddK (numV 1) (doneK)))
```

Exemples

```
(define (interp e env k)
  (type-case Exp e
    ...
    [(numE n) (continue k (numV n))]
    [(plusE l r)
     (interp l env (addSecondK r env k))]
    ... ))
```

```
(define (continue k val)
  (type-case Cont k
    ...
    [(doneK) val]
    [(addSecondK r env next-k)
     (interp r env (doAddK val next-k))]
    [(doAddK v-l next-k)
     (continue next-k (num+ v-l val))]
    ... ))
```

```
(interp (numE 2) mt-env
        (doAddK (numV 1) (doneK)))
```

➡ (continue (doAddK (numV 1) (doneK))
 (numV 2))

➡ (continue (doneK) (numV 3))

➡ (numV 3)

Interpréter avec des continuations

□ Cas de la lambda-abstraction

```

(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(idE s) (continue k (lookup s env))]
    ... ))

(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [ ... ]
    [ ...
      ...
      ...
      ... ]
    ... ))

```

Interpréter avec des continuations

□ Cas de la lambda-abstraction

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(lamE par body) (continue k (closV par body env))]
    ... ))
```

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [ ... ]
    [ ...
      ...
      ...
      ... ]
    ... ))
```


Interpréter avec des continuations

□ Cas de l'application

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(appE f arg) (interp f env (appArgK arg env k))]
    ... ))
```

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(appArgK arg env next-k) (interp arg env (doAppK val next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV par body c-env)
        [(closV par body c-env)
         (interp body (extend-env (bind par val) c-env) next-k)]
        [else (error 'interp "not a function")]]
       ... ))])
```

Avez-vous remarqué le léger changement de sémantique ?

Boucles infinies

- ❑ Que se passe-t-il en Java si l'on exécute la méthode suivante ?

```
public static void f() { while (true) {} }
```

- ❑ On obtient une boucle infinie.

- ❑ Et si on exécute ce code-ci ?

```
public static void g() { g(); }
```

- ❑ On obtient une boucle infinie... qui s'arrête très rapidement sur une **StackOverflowError**.

- ❑ Deux types de boucles, celles qui sont en **espace borné** et celles qui ne le sont pas.

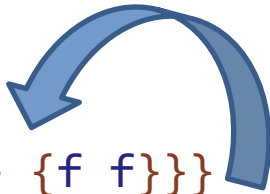
Et pour notre langage?

❑ Est-ce que l'expression suivante est une boucle en espace borné ?

```
{let {[f {lambda {f} {f f}}]}  
  {f f}}
```

❑ Que se passe-t-il en terme de substitution ?

```
{let {[f {lambda {f} {f f}}]}  
  {f f}}
```



```
{{lambda {f} {f f}} {lambda {f} {f f}}}
```

❑ On peut toujours représenter les appels successifs de manière concise, c'est une boucle en espace borné.

Evaluation avec les continuations

❑ Que se passe-t-il maintenant en terme de continuation ?

```
(interp {{lambda {f} {f f}}} {lambda {f} {f f}}} mt-env (doneK))
```

➡ (interp {lambda {f} {f f}}} mt-env (appArgK {lambda {f} {f f}}} mt-env (doneK)))

➡ (continue (appArgK {lambda {f} {f f}}} mt-env (doneK)) (closV 'f {f f} mt-env))

➡ (interp {lambda {f} {f f}}} mt-env (doAppK (closV 'f {f f} mt-env) (doneK)))

➡ (continue (doAppK (closV 'f {f f} mt-env) (doneK)) (closV 'f {f f} mt-env))

➡ (interp {f f} E (doneK))

E = (extend-env (bind 'f (closV 'f {f f} mt-env)) mt-env)

Evaluation avec les continuations

❑ Que se passe-t-il maintenant en terme de continuation ?

```
(interp {f f} E (doneK))
E = (extend-env (bind 'f (closV 'f {f f} mt-env)) mt-env)
```

➡ (interp f E (appArgK f E (doneK)))

➡ (continue (appArgK f E (doneK)) (closV 'f {f f} mt-env))

➡ (interp f E (doAppK (closV 'f {f f} mt-env) (doneK)))

➡ (continue (doAppK (closV 'f {f f} mt-env) (doneK)) (closV 'f {f f} mt-env))

➡ (interp {f f} E (doneK))

Et pour notre langage ?

❑ Est-ce que l'expression suivante est une boucle en espace borné ?

```
{let {[f {lambda {f} {+ 1 {f f}}}]
      {f f}}}
```

➡ `{{lambda {f} {+ 1 {f f}}} {lambda {f} {+ 1 {f f}}}}`

➡ `{+ 1 {{lambda {f} {+ 1 {f f}}} {lambda {f} {+ 1 {f f}}}}`

➡ `{+ 1 {+ 1 {{lambda {f} {+ 1 {f f}}} {lambda {f} {+ 1 {f f}}}}}`

...

❑ De plus en plus de calculs sont mis en attente, cette boucle n'est pas en espace borné.

Récursion terminale

```
(define (forever x)  
  (forever (not x)))
```

- L'appel de la fonction `forever` est un **appel terminal**, il n'y a aucune opération à effectuer après l'évaluation de `forever`.

```
(define (run-out-of-memory x)  
  (not (run-out-of-memory x)))
```

- L'appel de la fonction `run-out-of-memory` n'est pas un appel terminal, il reste des opérations à effectuer après son évaluation.

Récursion terminale

```
(define (forever x)
  (if x (forever #f) (forever #t)))
```

- ❑ Ici `forever` se trouve à l'intérieur d'une clause `if`, mais il n'y a toujours pas d'opération à effectuer après son évaluation.
- ❑ L'appel de fonction est encore une fois **terminal**.

```
(define (run-out-of-memory x)
  (if (run-out-of-memory x) #t #f))
```

- ❑ Ici, l'appel de la fonction `run-out-of-memory` n'est pas un appel terminal, il reste à évaluer une des branche du `if` après son évaluation.

Dans notre interpréteur

- ☐ La fonction `interp` appelle `interp` et `continue` uniquement en position terminale.
- ☐ La fonction `continue` appelle `interp` et `continue` uniquement en position terminale.
- ☐ La fonction `lookup` appelle `lookup` uniquement en position terminale.
- ☐ Ce sont les seules fonctions récursives.

- ☐ La continuation de notre interpréteur en plait ne grandit pas beaucoup.

Influence du langage d'implémentation

- Lorsqu'on était passé de l'interpréteur `ordresup.rkt` à l'interpréteur `ordresup-lazy.rkt`, on pouvait interpréter plus d'expressions de *notre* langage.

```
{{lambda {x} 1} {2 3}}
```

- Que se passe-t-il si l'on fait de même avec l'interpréteur `lambda-k.rkt` ?



lambda-k.rkt

- Il n'y a pas de différences, en prenant la main sur les continuations, on impose l'ordre d'évaluation quelque soit celui du langage d'implémentation.

ERREURS ET CONTEXTE

Implémenter les erreurs

- ❑ A l'heure actuelle, lorsque nous interprétons un programme erroné, nous utilisons la gestion d'erreurs du *langage d'implémentation*.

```
(test/exn (interp {+ 1 {2 3}}) "not a function")
```

- ❑ On souhaiterait gérer les erreurs d'évaluation directement dans l'interpréteur.

```
(test (interp {+ 1 {2 3}}) (errorV "not a function"))
```

Implémenter les erreurs

□ Pour implémenter les erreurs,

- On ajoute la variante `errorV` au type `Value`.

```
(define-type Value
```

```
...
```

```
[errorV (msg : String)])
```

- Et on modifie les appels à la fonction `error` de plait en renvoyant une `errorV` à la place.

```
(define (lookup [n : Symbol] [env : Env]) : Value
```

```
(cond
```

```
[(empty? env) (errorV "free identifier")]
```

```
[(equal? n (bind-name (first env))) (bind-val (first env))]
```

```
[else (lookup n (rest env))]))
```

Tests

❑ Désormais, on n'a plus besoin d'utiliser `test/exn` pour nos tests.

```
(test (interp (parse `{1 2}) mt-env (doneK)) (errorV "not a function"))
(test (interp (parse `{+ {lambda {x} x} 1}) mt-env (doneK)) (errorV "not a number"))
(test (interp (parse `y) mt-env (doneK)) (errorV "free identifier"))
```

❑ Qu'en est-il de ces tests ?

```
(test (interp (parse `{f 2}) mt-env (doneK)) (errorV "free identifier"))
--> (errorV "not a function")
(test (interp (parse `{{+ {lambda {x} x} 1} y}) mt-env (doneK))
      (errorV "not a number"))
--> (errorV "not a function")
```

❑ Les valeurs `errorV` se propagent à travers les continuations : c'est l'erreur la plus 'haute' qui est renvoyée et non la plus 'profonde'.

Implémenter les erreurs

- ❑ La solution est de permettre aux fonctions qui peuvent engendrer des erreurs de ne pas poursuivre la continuation.
- ❑ Une erreur interrompt le cours normal du programme.

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(idE s) (lookup s env k)]
    ... ))
```

C'est désormais `lookup` qui est en charge de la continuation.

```
(define (lookup [n : Symbol] [env : Env] [k : Cont]) : Value
  (cond
    [(empty? env) (errorV "free identifier")]
    [(equal? n (bind-name (first env))) (continue k (bind-val (first env)))]
    [else (lookup n (rest env) k)]))
```

Et en cas d'erreur, elle ne la poursuit pas.

Implémenter les erreurs

- ❑ La solution est de permettre aux fonctions qui peuvent engendrer des erreurs de ne pas poursuivre la continuation.
- ❑ Une erreur interrompt le cours normal du programme.

```
(define (continue [k : Cont] [val : Value]) : Value
```

```
(type-case Cont k
```

```
...
```

```
[(doAddK v-l next-k) (num+ v-l val next-k)]
```

```
[(doMultK v-l next-k) (num* v-l val next-k)]
```

```
... ))
```

De même `num-op` est en charge de la continuation.

```
(define (num-op [op : (Number Number -> Number)]
```

```
[l : Value] [r : Value] [k : Cont]): Value
```

```
(if (and (numV? l) (numV? r))
```

```
(continue k (numV (op (numV-n l) (numV-n r))))
```

```
(errorV "not a number"))
```

Et en cas d'erreur, elle ne la poursuit pas.

Implémenter les erreurs

- ❑ La solution est de permettre aux fonctions qui peuvent engendrer des erreurs de ne pas poursuivre la continuation.
- ❑ Une erreur interrompt le cours normal du programme.

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV par body c-env)
        (interp body (extend-env (bind par val) c-env) next-k)]
       [else (errorV "not a function")]])
    ... ))
```

Dans le cas de l'application de fonction, il n'y a rien à changer. On interrompait déjà la continuation.

Attraper les erreurs

❑ Dans n'importe quel langage, des cas exceptionnels peuvent survenir.

```
(/ 1 0) --> division by zero
```

❑ Dans la plupart d'entre eux, il existe des moyens de gérer ces cas exceptionnels.

```
(try (/ 1 0) (lambda () +inf.0)) --> +inf.0
```

❑ Nous allons implémenter un moyen de gérer les erreurs dans notre langage.

Attraper les erreurs : exemples

- Une expression `try` renvoie la valeur de l'expression testée si aucune erreur ne se produit.

```
(try (+ 0 1) (lambda () 2))  
--> 1
```

- Une expression `try` appelle son gestionnaire d'erreur dès que l'expression testée génère une erreur même dans une sous-expression.

```
(try (list 1 (/ 1 0) 3) (lambda () empty))  
--> empty
```

- On peut utiliser la valeur renvoyée par une expression `try` de manière classique.

```
(cons 4 (try (list 1 (/ 1 0) 3) (lambda () empty)))  
--> (cons 4 empty)
```

Attraper les erreurs : exemples

- ❑ On peut imbriquer les expressions `try`, c'est alors le plus 'proche' de l'erreur qui appelle son gestionnaire d'erreur.

```
(try (try (list 1 (/ 1 0) 3)
        (lambda () empty))
     (lambda () (list 4)))
--> empty
```

- ❑ Mais le gestionnaire d'erreur peut lui-même générer une erreur que l'on souhaite attraper.

```
(try (try (list 1 (/ 1 0) 3)
        (lambda () (/ 1 0)))
     (lambda () (list 4)))
--> (list 4)
```

Grammaire du langage

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {try <Exp> {lambda {} <Exp>}}
  
```

New !

```

(test {try 0 {lambda {} 1}}
  (numV 0))
  
```

Grammaire du langage

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {try <Exp> {lambda {} <Exp>}}
```

New !

```
(test {try {0 0} {lambda {} 1}}
      (numV 1))
```

Grammaire du langage

```
<Exp> ::= <Number>  
        | <Symbol>  
        | {+ <Exp> <Exp>}  
        | {* <Exp> <Exp>}  
        | {lambda {<Symbol>} <Exp>}  
        | {<Exp> <Exp>}  
        | {try <Exp> {lambda {} <Exp>}}
```

New !

```
(test {+ {try 0 {lambda {} 1}} 2}  
      (numV 2))
```

Grammaire du langage

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {try <Exp> {lambda {} <Exp>}}
```

New !

```
(test {+ {try {0 0} {lambda {} 1}} 2}
      (numV 3))
```


Grammaire du langage

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {try <Exp> {lambda {} <Exp>}}
  
```

New !

```

(test {try {try {0 0}
               {lambda {} 1}}}
      {lambda {} 2}}
(numV 1))
  
```

Grammaire du langage

```

<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {try <Exp> {lambda {} <Exp>}}
  
```

New !

```

(test {try {try {0 0}
                {lambda {} {1 1}}}}
      {lambda {} 2})
(numV 2))
  
```

Implémentation

□ On dispose d'un nouveau type d'expression dans notre langage.

```
(define-type Exp  
  ...  
  [tryE (body : Exp) (handler : Exp)]  
  ... )
```

```
(test (parse '{try {+ 1 2} {lambda {} 3}})  
      (tryE (addE (numE 1) (numE 2))  
            (numE 3)))
```

Notez que le `lambda` fait partie de la syntaxe de `try`.

Implémentation

□ On a aussi un nouveau type de continuation.

```
(define-type Cont  
  ...  
  [tryK (handler : Exp) (env : Env) (k : Cont)]  
  ... )
```

□ À l'évaluation, on interprète le corps du `try` et on met en attente le gestionnaire d'erreur.

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value  
  (type-case Exp e  
    ...  
    [(tryE body handler) (interp body env (tryK handler env k))]))
```

Implémentation

- ❑ Et lors de l'appel de la continuation, on appelle la continuation suivante puisque une valeur valide a été obtenue.

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(tryK handler env next-k) (continue next-k val)]
    ... ))
```

- ❑ Mais alors à quoi sert l'expression `try` puisqu'elle est transparente ?
- ❑ Elle n'intervient que dans la gestion d'erreurs et les erreurs court-circuitent le flot normal d'exécution.

Lancer des erreurs

- ❑ En cas d'erreur, au lieu de simplement renvoyer une `errorV`, il faut chercher si un `try` est en attente dans la continuation courante.

```
(errorV "not a number")
```



```
(escape k (errorV "not a number"))
```

Lancer des erreurs : exemples

- ❑ En cas d'erreur, au lieu de simplement renvoyer une `errorV`, il faut chercher si un `try` est en attente dans la continuation courante.
- ❑ Si aucun `try` n'est présent dans la continuation, on renvoie l'erreur elle-même.

```
(test (escape (doAddK (numV 1) (doneK))  
            (errorV "fail"))  
      (errorV "fail"))
```

Continuation courante

Erreur courante

Lancer des erreurs : exemples

- ❑ En cas d'erreur, au lieu de simplement renvoyer une `errorV`, il faut chercher si un `try` est en attente dans la continuation courante.
- ❑ Si un `try` est présent dans la continuation, on exécute son gestionnaire d'erreur.

```
(test (escape (doAddK (numV 1)
                     (tryK (numE 2) mt-env (doneK))))
      (errorV "fail"))
(numV 2))
```


Lancer des erreurs : exemples

- ❑ En cas d'erreur, au lieu de simplement renvoyer une `errorV`, il faut chercher si un `try` est en attente dans la continuation courante.
- ❑ Si un `try` est présent dans la continuation, on exécute son gestionnaire d'erreur et on reprend la continuation à ce moment.

```
(test (escape (doAddK (numV 1)
                      (tryK (numE 2) mt-env
                            (doAddK (numV 3) (doneK))))))
      (errorV "fail"))
(numV 5))
```

Implémentation

```
(define (escape [k : Cont] [val : Value]) : Value
  (type-case Cont k
    [(doneK) val]
    [(addSecondK r env next-k) (escape next-k val)]
    [(doAddK v-l next-k) (escape next-k val)]
    [(multSecondK r env next-k) (escape next-k val)]
    [(doMultK v-l next-k) (escape next-k val)]
    [(appArgK arg env next-k) (escape next-k val)]
    [(doAppK v-f next-k) (escape next-k val)]
    [(tryK handler env next-k) (interp handler env next-k)]))
```

Saut de contexte

□ L'expression `try` permet de passer du contexte courant à un contexte antérieur si une erreur est lancée.

□ En évaluant

```
(+ 1 (try (+ 2 (+ 3 (+ 4 (5 6)))))  
      (lambda () (+ 7 8))))
```

on passe du contexte où l'on évalue `(5 6)` avec les différentes additions en attente au contexte `(+ 1 •)` avec la valeur 15.

□ Rétablir un contexte est un mécanisme plus général.

Saut de contexte

- La forme `let/cc` permet de sauvegarder le contexte courant et de le lier à un identificateur pour le restaurer ultérieurement.

```
(+ 1
  (let/cc k1
    (+ 2
      (+ 3
        (let/cc k2
          (+ 4
            (k1 5)))))))
```

permet de rétablir le contexte `(+ 1 •)` avec la valeur 5.

Saut de contexte

□ La forme `let/cc` permet de sauvegarder le contexte courant et de le lier à un identificateur pour le restaurer ultérieurement.

```
(+ 1
  (let/cc k1
    (+ 2
      (+ 3
        (let/cc k2
          (+ 4
            (k2 5)))))))
```

permet de rétablir le contexte `(+ 1 (+ 2 (+ 3 •)))` avec la valeur 5.

Saut de contexte

```
(define continue (lambda (n) n))
```

```
(let/cc return
```

```
  (+ 1
```

```
    (+ 2
```

```
      (+ 3
```

```
        (+ 4
```

```
          (let/cc k
```

```
            (begin
```

```
              (set! continue k)
```

```
              (return 5)))))))))
```

```
(continue 6)
```

On capture le contexte `(+ 1 (+ 2 (+ 3 (+ 4 •))))`
et on le sauvegarde

Saut de contexte

```

(define continue (lambda (n) n))

(let/cc return
  (+ 1
    (+ 2
      (+ 3
        (+ 4
          (let/cc k
            (begin
              (set! continue k)
              (return 5))))))))))

```

Cela revient à dire que la variable `continue` contient la fonction `(lambda (v) (+ 1 (+ 2 (+ 3 (+ 4 v)))))`

D'ailleurs, quelle est la valeur de cette expression?

`(continue 6)` → On peut appeler cette continuation à volonté.

Grammaire du langage avec `let/cc`

```
<Exp> ::= <Number>  
      | <Symbol>  
      | {+ <Exp> <Exp>}  
      | {* <Exp> <Exp>}  
      | {lambda {<Symbol>} <Exp>}  
      | {<Exp> <Exp>}  
      | {let/cc <Symbol> <Exp>}
```

New !

Continuations en tant que valeurs

- La forme `let/cc` lie la continuation courante à un identificateur, les continuations se trouvent donc dans l'environnement et doivent être des valeurs.

```
(define-type Value  
  ...  
  [contV (k : Cont)])
```

Continuations en tant que valeurs

- ❑ Evaluer `let/cc` revient à évaluer son corps dans un environnement enrichi de la continuation courante.

```
(define (interp [e : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp e
    ...
    [(let/ccE s body) (interp body (extend-env (bind s (contV k)) env) k)]
    ... ))
```

Continuations en tant que valeurs

- ❑ Les continuations s'utilisent comme des fonctions, il faut ajouter un cas quand on traite une continuation `doAppK`.
- ❑ Dans le cas où le premier argument est une continuation, il suffit de la rétablir et de continuer le calcul.

```
(define (continue [k : Cont] [val : Value]) : Value
  (type-case Cont k
    ...
    [(doAppK (v-f next-k)
      (type-case Value v-f
        [(closV par body c-env) ... ]
        [(contV k-v) (continue k-v val)]
        [else ... ])]
    ... ))
```

Exemples

❑ Essayez de trouver la valeur des expressions suivantes :

```
{let {[f {let/cc k k}]}  
  {f {lambda {x} 1}}}
```

❑ Le contexte sauvegardé dans `k` est :

```
{let {[f •]}  
  {f {lambda {x} 1}}}
```

❑ Ce contexte est lié à l'identificateur `f` puis appelé avec `{lambda {x} 1}` comme argument. Cela revient donc à évaluer :

```
{let {[f {lambda {x} 1}]}  
  {f {lambda {x} 1}}}
```

Exemples

❑ Essayez de trouver la valeur des expressions suivantes :

```
{{let/cc esc
  {let {[v {+ 1 {let/cc k {esc k}}}]
    {lambda {x} {+ x v}}}}
  2}}
```

❑ Le contexte sauvegardé dans `esc` est `{• 2}` et celui dans `k` est :

```
{{let {[v {+ 1 •}]}
  {lambda {x} {+ x v}}}
  2}}
```

❑ L'expression évaluée est `{esc k}`, c'est-à-dire `{k 2}`, ou encore :

```
{{let {[v {+ 1 2}]}
  {lambda {x} {+ x v}}}
  2}}
```

Utilisation des continuations

- De manière générale, les continuations sont peu utilisées explicitement dans les programmes.

- Elles servent surtout à implémenter d'autres concepts avancés :
 - Gestionnaire d'exceptions
 - Threads
 - Générateurs
 - ...

Threads

```
(define (count label n)
  (begin
    (sleep)
    (display label)
    (display (to-string n))
    (display "\n")
    (count label (+ n 1))))
```

On endort le processus courant.
Notez l'absence d'argument.

Création d'un processus à partir d'une fonction.

```
(thread (lambda (vd) (count "a" 0)))
(thread (lambda (vd) (count "b" 0)))
(run)
```

Lancement des processus existants

Threads

```
(define threads empty)
```

Liste des processus

```
(define (thread k)  
  (set! threads (append threads (list k))))
```

Ajout d'un processus à la liste

```
(define (run)  
  (let ([f (first threads)])  
    (begin  
      (set! threads (rest threads))  
      (f (void))))))
```

On sort et on récupère le premier processus de la liste.

Et on l'exécute

```
(define (sleep)  
  (let/cc k  
    (begin  
      (thread k)  
      (run))))
```

Pour mettre un processus en pause, on capture sa continuation (ce qu'il reste à faire) puis on crée un nouveau processus dont la fonction est cette continuation pour reprendre au point d'arrêt.

Générateurs

```
(define (make-numbers start-n)
  (generator yield
    (local [(define (numbers n)
              (begin
                (yield n)
                (numbers (+ n 1))))])
    (numbers start-n))))
```

```
(define g (make-numbers 0))
```

```
(g) --> 0
```

```
(g) --> 1
```

```
(g) --> 2
```

```
...
```

On crée un générateur à partir d'un nom d'échappement (ici `yield`) et d'un corps.

À chaque fois que `yield` apparaît, le générateur met en pause l'évaluation du corps et renvoie la valeur passée en paramètre.

Générateurs

```
(define-syntax-rule (generator yield-id body)
  (make-generator (lambda (yield-id)
                    (lambda (vd)
                      body))))
```

Macro pour introduire le nom d'échappement.

```
(define (make-generator proc)
  (local [(define yield identity)
          (define go (proc (lambda (v) (yield v))))])
```

Variable qui contient le corps à exécuter, on lie le vrai `yield` au nom dans la macro.

```
(lambda ()
```

Point de retour de `yield`.

```
(let/cc escape
```

```
(begin
```

```
(set! yield (lambda (v)
```

Quand `yield` est appelé, on sauvegarde le contexte et on s'échappe.

```
(let/cc k
```

```
(begin (set! go k) (escape v))))
```

```
(go (void))))))
```

On lance le générateur.