Implantation de la récursion

Motivation

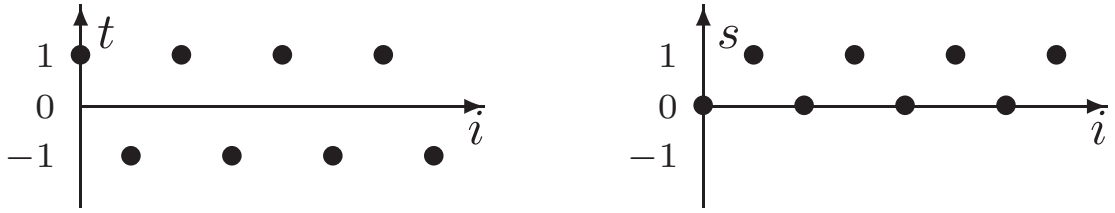
Implantons la fonction suivante:

$$f : \mathbb{N} \rightarrow \mathbb{N}$$
$$f(n) = \sum_{i=0}^{n-1} (-1)^i$$

en Scheme:

```
(define (f n) (f_aux n 0 1 0))
(define (f_aux n i t s)
  (if (= i n)
      s
      (f_aux n (+ i 1) (- t) (+ s t))))
```

Graphiquement:



Étant donnés les nombres manipulés par cette fonction, il est raisonnable de vouloir effectuer un calcul comme `f 100000000`.

Motivation

Traduction directe en C:

```
int f(int n) { return f_aux(n, 0, 1, 0); }
int f_aux(int n, int i, int t, int s)
{
  if (i == n)
    return s;
  else
    return f_aux(n, i + 1, - t, s + t);
}
```

`f(100000000)` va faire déborder la pile d'à peu près n'importe quelle implantation.

Observation: quand `f_aux` fait un appel à `f_aux`, il est inutile de retourner à l'activation appelante, puisque celle-ci ne fait que retourner immédiatement le même résultat. Le résultat final pourrait être retourné directement à `f`.

Position terminale vs. non-terminale

Définitions:

- Une expression est en *position terminale* si la valeur qu'elle fournit est retournée directement par la fonction qui l'englobe.
- Une expression est en *position non-terminale* si la valeur qu'elle fournit doit être manipulée de quelque façon que ce soit par la fonction qui l'englobe ou bien elle se trouve au niveau global.
- Un *appel terminal* est un appel de fonction en position terminale.
- Un *appel non-terminal* est un appel de fonction en position non-terminale.

Position terminale vs. non-terminale

Exemples:

Soit e une expression. Elle est soit en position terminale, soit en position non-terminale. Qu'en est-il de ses sous-expressions?

- $e =$ `(set! x` e_1`)`
- $e =$ `(if` e_1 e_2 e_3`)`
- $e =$ `(let ((x` e_1`))` e_2`)`
- $e =$ `(lambda (x)` e_1`)`
- $e =$ `(`f e_1 `...` e_n`)`

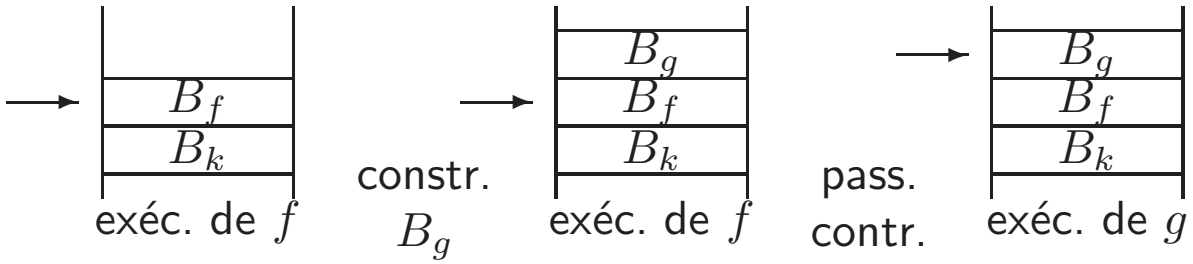
Réponses:
[e_1 non] [e_1 non; e_2 et e_3 comme e] [e_1 non; e_2 comme e] [e_1 oui] [f, e_1, ..., e_n non]

Importance de l'implantation de la récursion

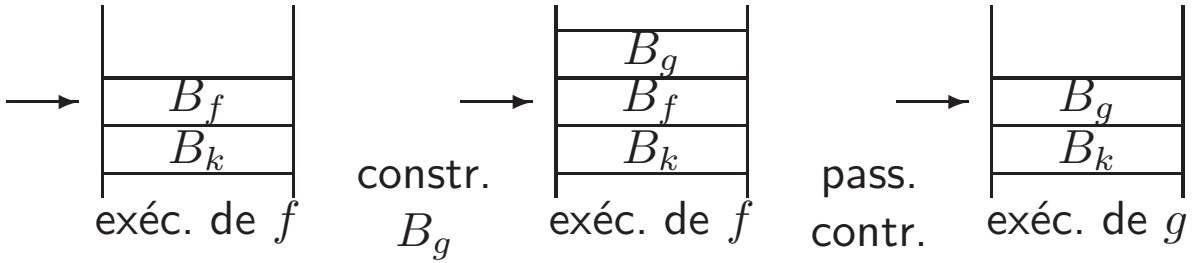
À cause de l'importance des appels et de la récursion, l'implantation devrait effectuer les appels terminaux de façon spéciale:

> après que la fonction f ait préparé l'appel à la fonction g (bloc d'activation B_g) et au moment où le contrôle est passé à la fonction g, le bloc d'activation B_f de f devrait être éliminé.

Graphiquement, au lieu de:



on veut que:



Importance de l'implantation de la récursion

Implantation (simpliste mais) correcte en C de la récursion terminale de `f_aux`:

```
int f(int n) { return f_aux(n, 0, 1, 0); }
int f_aux(int arg_n, int arg_i, int arg_t, int arg_s)
{
  int n, i, t, s;
  n = arg_n; i = arg_i; t = arg_t; s = arg_s;
  while (1)
    {
      if (i == n)
        return s;
      else
        {
          arg_n = n; arg_i = i + 1; arg_t = - t; arg_s = s + t;
          /* Appel terminal virtuel */
          n = arg_n; i = arg_i; t = arg_t; s = arg_s;
        }
    }
}
```

Capable d'exécuter `f(100000000)`.

Importance de l'implantation de la récursion

Considérations:

- Étant donnée l'importance de la récursion dans les langages fonctionnels, une implantation a intérêt à implanter les appels terminaux spécialement.
- Étant donnée la présence de l'implantation spéciale de la récursion terminale, le programmeur a intérêt à s'en servir lorsque c'est possible.
- Cette dernière peut rendre un calcul faisable à l'intérieur de la mémoire disponible.
- Elle peut rendre un calcul plus efficace en espace (e.g. $O(n) \mapsto O(1)$ pour `f_aux`) et en temps (moins de blocs d'activation et de données inutiles sont retenus vivants).

Exemple de code à éviter pour `f`, même en Scheme:

```
(define (f n) (f_aux n 0 1))
(define (f_aux n i t)
  (if (= i n)
      0
      (+ t (f_aux n (+ i 1) (- t)))))
```

Objectif

- On veut que tout appel terminal cause l'abandon du bloc d'activation de la fonction appelante.
- Le truc employé dans notre traduction simpliste en C pourrait être utilisé dans les situations où la fonction est auto-récursive.
- Il nous faut une technique qui fonctionne en général: fonctions mutuellement récursives, mélange d'appels terminaux et non-terminaux.

Nous allons nous intéresser à quelques méthodes.

Méthode “trampoline”

Implantée à l'aide d'une transformation source à source qui laisse le programme dans un état où:

- une λ-expression pourra être traduite directement en une fonction C (un appel en Scheme restera un appel en C);
- le programme transformé fait des récursions aussi profondément que ce que le programme original faisait à travers ses appels *non-terminaux*.

Comportement du programme original par rapport à celui du programme transformé:

- retour de valeur chez l'original $\mapsto$ retours chez le transformé;
- appel non-terminal chez l'original $\mapsto$ appels (en nombre constant) chez le transformé;
- appel terminal chez l'original $\mapsto$ retours suivi d'appels (en nombre égal) chez le transformé.

Donc, après traduction du programme transformé en C, l'utilisation de la pile C est restreinte “à ce qui est vraiment nécessaire”.

Méthode “trampoline”

La gestion du passage et du retour des valeurs va être gérée manuellement dans le programme transformé.

Hypothèses de travail:

- Nous illustrons la méthode trampoline en ne considérant que des fonctions sans variables libres.
- Nous utiliserons quelques fonctions auxiliaires, lesquelles seront présentées plus loin.
- L'implantation du programme transformé pourra paraître coûteuse mais pratiquement tous les coûts supplémentaires pourraient être éliminés par *inlining* et spécialisation du code généré.

Méthode “trampoline”

Transformation de la fonction:

```
(define (f a b c)
  (if b a c))                          ; retour de valeur
```

en:

```
(define (f a b c)
  (if b
      (TNTreturn a)
      (TNTreturn c)))                  ; fct. aux. de retour
```

Méthode “trampoline”

Transformation de la fonction:

```
(define (g a b)
  (h b 18 a))                          ; appel terminal
```

en:

```
(define (g a b)
  (TNTterm-apply h b 18 a))            ; fct. aux. d'appel term.
```

Méthode “trampoline”

Transformation de la fonction:

```
(define (h a b c)
  (or (i c b)                          ; appel non-terminal
      42))
```

en:

```
(define (h a b c)
  (or (TNTnon-term-apply i c b)    ; fct. aux. d'appel n.-t.
      (TNTreturn 42)))
```


Méthode “trampoline”

Fonctions utilitaires:

```
(define (TNTreturn x)
  (set! TNTtrue-return? #t)
  x)

(define (TNTterm-apply f . args)
  (set! TNTtrue-return? #f)
  (set! TNTop f)
  (set! TNTargs args))

(define (TNTnon-term-apply f . args)
  (set! TNTop f)
  (set! TNTargs args)
  (TNTspin))

(define (TNTspin)
  (do-until                                    ; forme SPECIALE interne
    (set! TNTval (apply TNTop TNTargs))        ; corps
    TNTtrue-return?)                           ; condition
  TNTval)
```

Méthode “trampoline”

Exemple: Fonction d’Ackermann:

```
(define (ack m n)
  (if (= m 0)
      (+ n 1)
      (if (= n 0)
          (ack (- m 1) 1)
          (ack (- m 1) (ack m (- n 1))))))
```

Transformée: ...

Méthode des blocs d'activation explicites

Remplaçons l'utilisation de la récursivité en C par une gestion explicite (dans le tas) des blocs d'activation.

Avantage:

+ Le programme résultant est complètement indépendant de la pile C.

Allure du programme C généré:

- une λ-expression produit une suite d'énoncés C;
- le code C résultant ne comporte aucune récursion;
- une seule fonction C contient les énoncés de toutes les λ-expressions et du programme principal;
- des énoncés "`goto`s" sont utilisés pour simuler les appels *et* les retours du programme fonctionnel;
- un bloc d'activation contient essentiellement les informations nécessaires à la réactivation de la fonction appelante; entre autres, on y retrouve l'adresse du code où le contrôle doit être retourné ainsi que les variables nécessaires à la réactivation.

Méthode des blocs d'activation explicites

Hypothèses de travail supplémentaires:

- Nous supposons que nous disposons du compilateur C `gcc`, lequel permet de manipuler l'adresse des étiquettes.
  – `etiq:` déclare une étiquette où on peut brancher;
  – `goto etiq;` saute directement à l'étiquette;
  – `&&etiq` est une expression de type `void *` qui produit l'adresse de l'étiquette;
  – `goto *<exp>` saute à l'adresse *calculée* par `<exp>`.

- On dispose d'un type `BLOC` qui est une structure de données capable de sauvegarder n'importe quel bloc d'activation.

La mécanique d'appel ainsi que celle des variables locales utilisent les variables globales C suivantes:

```
void *fct2call;                 /* fct. a appeler */
DATA arg1, arg2, ..., argn;     /* var. de transit pour les arg. */
DATA ret_val;                   /* var. de transit pour le ret. */
BLOC *bp;                       /* ptr. du bloc d'act. sauve */
DATA px, py, pz, ...;           /* var. contenant les param. */
```

Méthode des blocs d'activation explicites

Compilation de fonction en cas de retour de valeur:

```
(define (f x y z)
  (+ x z))                          ; Supposons que + est un operateur
```

en les énoncés C:

```
LAM12:                              /* point d'entree */
  px = arg1; py = arg2; pz = arg3;  /* reception des arguments */
  ret_val = px + pz;                /* valeur de retour */
  goto do_return;                   /* saut a la mecanique de ret. */
```

Méthode des blocs d'activation explicites

Compilation de fonction en cas d'appel terminal:

```
(define (g x y)
  (h y 18 x))
```

en les énoncés C:

```
LAM37:
  px = arg1; py = arg2;
  fct2call = &&LAM41;              /* preparation de ... */
  arg1 = py; arg2 = 18; arg3 = px;   /* ... l'appel */
  goto do_invoke;                  /* saut a la mecanique d'invoc. */
```

Méthode des blocs d'activation explicites

Compilation de fonction en cas d'appel non-terminal:

```
(define (h x y z)
  (+ 1 x (i z y)))                    ; Supposons que + est un operateur
```

en les énoncés C:

```
LAM41:
  px = arg1; py = arg2; pz = arg3;
  fct2call = &&LAM62;
  arg1 = pz; arg2 = py;
  bp = new BLOC(&&LAM42, bp, px);    /* bloc de sauvegarde */
  goto do_invoke;
LAM42:                               /* point de retour d'appel */
  px = ...; bp = ...;                /* retablissement de px et bp */
  ret_val = 1 + px + ret_val;
  goto do_return;
```

Méthode des blocs d'activation explicites

Mécaniques utilitaires:

- mécanique d'invocation:

```
do_invoke:
  goto *fct2call;
```

- mécanique de retour:

```
do_return:
  goto *get_return_point(bp);
```

Méthode des blocs d'activation explicites

Caractéristiques du code généré:

- Le bloc d'activation d'une fonction *en cours d'exécution* se trouve dans les variables globales.
- Le bloc d'activation de chaque fonction dans la chaîne des fonctions appelantes se trouve dans la chaîne de structures `BLOC` pointée par `bp`.
- Dès qu'elle est invoquée, une fonction commence par libérer les variables globales `arg*` (réception des arguments).
- Lorsqu'elle retourne directement une valeur, la fonction place cette valeur dans `ret_val` et passe le contrôle à `do_return`.
- Lorsqu'elle fait un appel terminal, la fonction prépare les variables `fct2call` et `arg*` et passe le contrôle à `do_invoke`.
- Lorsqu'elle fait un appel non-terminal, la fonction prépare les variables `fct2call` et `arg*` et *enrobe* le passage de contrôle à `do_invoke` avec la sauvegarde du bloc d'activation dans une structure `BLOC`; celle-ci inclut l'adresse du point de retour de l'appel; au retour, toutes les variables ont été écrasées, sauf `bp`, et leur valeur est sans signification, sauf pour (`bp` et) `ret_val`. On dit que `bp` est une variable *callee-save* et que les autres sont *caller-save*.

Méthode des blocs d'activation explicites

Il est facile d'adapter cette méthode pour permettre l'utilisation de fonctions comme objets de première classe.

1. La variable globale `fct2call` sert maintenant à contenir des fermetures:

   ```
   DATA fct2call;
   ```

2. À chaque préparation d'appel, `fct2call` doit être initialisée avec une fermeture (une donnée de type `DATA`) au même titre que les variables `arg*`.

3. La mécanique d'invocation devient:

   ```
   do_invoke:
     goto *get_clos_code(fct2call);
   ```

4. À chaque début de fonction, en plus de libérer les variables `arg*`, il faut libérer `fct2call` soit en copiant les variables capturées par la fermeture, soit en conservant une référence à la fermeture elle-même.

5. Une fermeture est une structure contenant les variables capturées et *une adresse d'étiquette*.

Méthode des blocs d'activation explicites

Exemple de programme à compiler:

```
(define (inc x)
  (+ x 1))
(define (comp f g)
  (lambda (x)
    (f (g x))))

((comp inc inc) 5)
```

Méthode des blocs d'activation explicites

Programme C équivalent avec représentation plate des fermetures:

```
int main(int argc, char *argv[])
{
  DATA fct2call, arg1, arg2, ret_val;    /* variables ... */
  BLOC *bp;                              /* ... utilitaires */
  DATA inc, comp;                        /* var. globales Scheme */
  DATA px, pf, pg;                       /* var. locales Scheme */

  bp = NULL;                             /* initialisation */
  inc = make_clos1_0(&&LAM1);            /* programme principal */
  comp = make_clos2_0(&&LAM2);
  fct2call = comp;                       /* prepare l'appel a comp */
  arg1 = inc; arg2 = inc;
  bp = new BLOC(&&RETPT1, bp);           /* pseudo-C */
  goto do_invoke;
RETPT1:
  bp = get_bp(bp);
  fct2call = ret_val;                    /* prepare l'appel a inc o inc */
  arg1 = make_int(5);
  bp = new BLOC(&&RETPT2, bp);           /* pseudo-C */
  goto do_invoke;
RETPT2:
  bp = get_bp(bp);
  return extract_int(ret_val);
```

à suivre ...

Méthode des blocs d'activation explicites

suite ...

```
LAM1:                                   /* fonction inc */
  px = arg1;
  ret_val = make_int(extract_int(px) + 1);
  goto do_return;

LAM2:                                   /* fonction comp */
  pf = arg1; pg = arg2;
  ret_val = make_clos1_2(&&LAM3, pf, pg);
  goto do_return;

LAM3:                                   /* lambda-expression */
  pf = get_clos_var1(fct2call); pg = get_clos_var2(fct2call); px = arg1;
  fct2call = pg;
  arg1 = px;
  bp = new BLOC(&&RETPT3, bp, pf);      /* pseudo-C */
  goto do_invoke;
RETPT3:
  pf = get_block_var1(bp);  bp = get_bp(bp);
  fct2call = pf;
  arg1 = ret_val;
  goto do_invoke;

do_invoke:                              /* mecanique utilitaire */
  goto *get_clos_code(fct2call);
do_return:
  goto *get_return_point(bp);
}
```

Méthode des blocs d'activation explicites

Remarque:

Les fermetures et les blocs d'activation sauvés (`BLOC`) sont similaires:

- ils contiennent tous l'adresse d'une étiquette;
- ils contiennent tous des valeurs à sauvegarder.

De plus, ils s'utilisent de façon presque similaire.

Idée: En les unifiant, on obtient la méthode suivante.

Méthode des blocs d'activation explicites

Il n'est pas absolument nécessaire d'utiliser l'extension des gotos calculés de gcc. On peut utiliser l'énoncé standard `switch` de C. Il y a toutefois une certaine perte d'efficacité.

```
>                                          > while (1)
>                                          > {
>                                          >   switch (dest)
>                                          >     {
>   ...                                    >       ...
>   ...                                    >       ...
> etiq12:                                  >     case 12:
>   ...                                    >       ...
>   ...                                  ⟼ >       ...
>   x = ... &&etiq12 ...;                  >       x = ... 12 ...;
>   ...                                    >       ...
>   ...                                    >       ...
>   goto exp; /* type void * */            >       dest = exp'; /* type int */
>                                          >       break;
>   ...                                    >       ...
>   ...                                    >       ...
>                                          >     }
>                                          > }
```

Méthode “conversion en forme CPS”

Définition: Un programme est en *forme CPS*, pour *continuation-passing style*, si la continuation de toute expression y est présente explicitement sous la forme d'une fonction à un argument et que tout retour de valeur se fait à l'aide d'un appel à cette dernière.

Cette méthode d'implantation de la récursion en est une de compilation en deux phases. La première phase consiste en la conversion en forme CPS, laquelle est une transformation source-à-source. La seconde consiste en une compilation en C à l'aide d'une des méthodes vues précédemment, i.e. la méthode “trampoline” ou la méthode des blocs d'activation explicites.

Caractéristiques du code produit par la conversion:

- les fonctions reçoivent un paramètre de plus: la continuation courante;
- à proprement parler, il n'y a plus de notion de “retour de valeur” effectué par les fonctions;
- il n'y a plus d'appels non-terminaux, seulement des appels terminaux.

Méthode “conversion en forme CPS”

Supposons que le langage que l'on veut implanter ait la syntaxe suivante:

e	::=	c	Constante
	|	x	Référence à une variable
	|	$\lambda \mathrm{x} \ldots \mathrm{x}.\ e$	λ-expression
	|	$e\ e \ldots e$	Appel de fonction
	|	$\mathrm{let}\ \mathrm{x} = e\ \mathrm{in}\ e$	Liaison d'une variable locale
	|	$\mathrm{if}\ e\ \mathrm{then}\ e\ \mathrm{else}\ e$	Conditionnelle
	|	$e + e$	Opération arithmétique d'addition
	|	$e : e$	Construction d'une paire
	|	$\#1\ e$	Extraction du champ d'une paire
	|	$\#2\ e$	

Hypothèse de travail:

- on suppose que le programme est α-converti, i.e. toutes les variables ont un nom distinct.

Méthode “conversion en forme CPS”

Règles de conversion en forme CPS. Un programme est converti en forme CPS à l'aide de la fonction $\mathrm{CPS\_main}$.

$$\mathrm{CPS\_main}[\![e]\!] = \mathrm{CPS}[\![e]\!][\![\mathcal{K}_0]\!]$$

où $\mathcal{K}_0$ est la *continuation finale* du programme.
Note: toutes les variables introduites par la transformation (y, y_0, ..., y_n, k) ont des noms distincts des variables déjà existantes.

$$\mathrm{CPS}[\![c]\!][\![\mathrm{K}]\!] = \\ \quad \mathrm{K}\ c$$

$$\mathrm{CPS}[\![\mathrm{x}]\!][\![\mathrm{K}]\!] = \\ \quad \mathrm{K}\ \mathrm{x}$$

$$\mathrm{CPS}[\![\lambda \mathrm{x}_1 \ldots \mathrm{x}_n.\ e]\!][\![\mathrm{K}]\!] = \\ \quad \mathrm{K}\ (\lambda \mathrm{x}_1 \ldots \mathrm{x}_n\ \mathrm{k}.\ \mathrm{CPS}[\![e]\!][\![\mathrm{k}]\!])$$

(à suivre ...)

Méthode “conversion en forme CPS”

(Suite . . .)

$$
\begin{aligned}
&\mathrm{CPS}[\![e_0\ e_1 \ldots e_n]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e_0]\!][\![\lambda \mathrm{y}_0. \\
&\qquad \mathrm{CPS}[\![e_1]\!][\![\lambda \mathrm{y}_1. \\
&\qquad\quad \ldots \\
&\qquad\qquad \mathrm{CPS}[\![e_n]\!][\![\lambda \mathrm{y}_n. \\
&\qquad\qquad\quad \mathrm{y}_0\ \mathrm{y}_1 \ldots \mathrm{y}_n\ \mathrm{K}]\!] \ldots]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CPS}[\![\mathrm{let\ x} = e_1\ \mathrm{in}\ e_2]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e_1]\!][\![\lambda \mathrm{x}. \\
&\qquad \mathrm{CPS}[\![e_2]\!][\![\mathrm{K}]\!]]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CPS}[\![\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e_1]\!][\![\lambda \mathrm{y.\ if\ y\ then\ CPS}[\![e_2]\!][\![\mathrm{K}]\!]\ \mathrm{else\ CPS}[\![e_3]\!][\![\mathrm{K}]\!]]\!]
\end{aligned}
$$

(à suivre . . .)

Méthode “conversion en forme CPS”

(Suite . . .)

$$
\begin{aligned}
&\mathrm{CPS}[\![e_1 + e_2]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e_1]\!][\![\lambda \mathrm{y}_1. \\
&\qquad \mathrm{CPS}[\![e_2]\!][\![\lambda \mathrm{y}_2. \\
&\qquad\quad \mathrm{cps\_plus\ y}_1\ \mathrm{y}_2\ \mathrm{K}]\!]]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CPS}[\![e_1 : e_2]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e_1]\!][\![\lambda \mathrm{y}_1. \\
&\qquad \mathrm{CPS}[\![e_2]\!][\![\lambda \mathrm{y}_2. \\
&\qquad\quad \mathrm{cps\_cons\ y}_1\ \mathrm{y}_2\ \mathrm{K}]\!]]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CPS}[\![\#1\ e]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e]\!][\![\lambda \mathrm{y}. \\
&\qquad \mathrm{cps\_car\ y\ K}]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{CPS}[\![\#2\ e]\!][\![\mathrm{K}]\!] = \\
&\quad \mathrm{CPS}[\![e]\!][\![\lambda \mathrm{y}. \\
&\qquad \mathrm{cps\_cdr\ y\ K}]\!]
\end{aligned}
$$

Méthode “conversion en forme CPS”

Exercice: convertir en forme CPS les expressions suivantes

$$(\lambda \mathrm{a\ b.\ a} + \mathrm{b})\ 5\ 8$$

$$\begin{array}{l}\mathrm{let\ lst} = 12 : [\,] \mathrm{\ in} \\ \quad \mathrm{if\ lst\ then\ \#1\ lst} \\ \qquad\quad \mathrm{else\ lst}\end{array}$$

Méthode “conversion en forme CPS”

Avantages et inconvénients:

+ Produit un programme beaucoup plus facile à compiler vers C.
- Produit du code moins efficace à moins que des optimisations ne soient faites afin de corriger la situation.
  - Plus d'utilisation d'une pile.
  - Introduction de nombreuses λ-expressions *administratives*.
  - Doit fixer l'ordre d'évaluation.
- Nécessite une α-conversion afin d'éviter la capture accidentelle de variables.