INF 544 Luc Maranget

Compilation

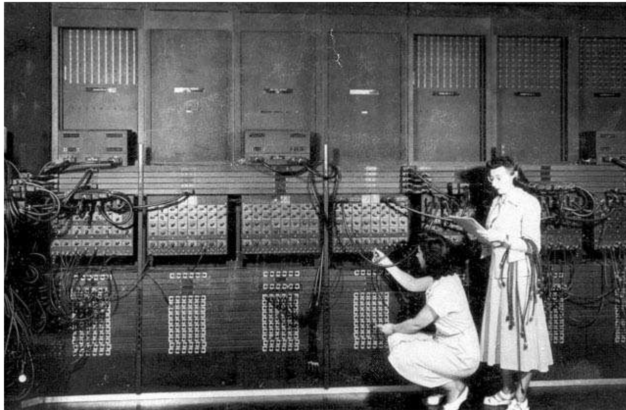
Luc.Maranget@inria.fr
http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/

A Qu'est-ce que compiler ?

B Machine abstraite.

C Compilation de PCF.

Presque un ordinateur



Programmer l'ENIAC (1946), c'est connecter des câbles.

Un peu plus tard (Iris 10 010, 1967)

Programmer un ordinateur, c'est entrer un programme en mémoire.

Un ordinateur

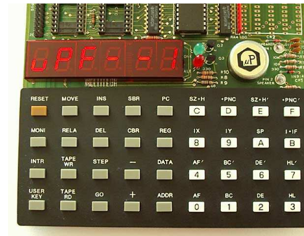
Par définition ou presque (machine de Von Neumann), un ordinateur est :

- Une machine « universelle ».
- Composée d'une unité de contrôle et d'une mémoire.
- L'UC lit (et exécute en séquence) un *programme* à partir de la mémoire.
- Le programme est une séquence d'instructions élémentaires :
  - Lire/Écrire une case de mémoire.
  - Additionner le contenu de deux cases de mémoire.
  - Lire (et exécuter) une instruction qui n'est pas la suivante.
  - etc.

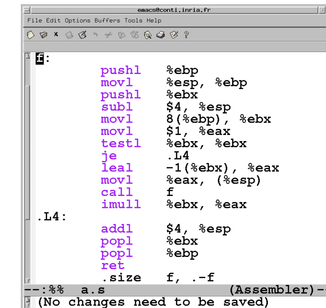
Entrer un programme en mémoire

Il s'agit d'un détail technique. Par exemple. . .

Années 80, « amateur » fauché



Aujourd'hui, assembleur



Et PCF ?

Il n'existe pas d'ordinateur qui exécutent les programmes PCF directement.

Car PCF s'addresse d'abord aux être humains (sisi).

Deux techniques pour exécuter PCF.

- Écrire un interpréteur, ce qui repousse le problème (dans quel langage écrire l'interpréteur ?).
- Traduire PCF dans le langage de la machine, c-a-d compiler.

Machine abstraite

Traduire PCF vers une vraie machine, c'est trop compliqué.

Car la vraie machine est un peu trop simple, et il y aura des complications pour les fonctions.

À la place, nous traduisons vers une machine *abstraite*. Puis. . .

- Interpréter la machine abstraite (comme Java ou `ocamlc`).
- Ou traduire le programme de la machine abstraite vers un programme machine.
  - Tout de suite (comme `ocamlopt`).
  - Lors de l'exécution (*Just In Time compilation*, comme Java).

Poule et œuf

Imaginons que PCF est le seul langage $\mathcal{L}$ (informatique) de l'humanité, qui vient d'inventer son premier ordinateur $\mathcal{M}$. Pour pouvoir exécuter un programme écrit en $\mathcal{L}$, il faut :

- Écrire un interpréteur I, de $\mathcal{L}$, dans le langage de $\mathcal{M}$.

Ensuite...

- On peut écrire un compilateur de $\mathcal{L}$ vers $\mathcal{M}$ en $\mathcal{L}$ (plus commode qu'en $\mathcal{M}$).
- Compiler le compilateur (grâce à I), ce qui donne C.
- On peut maintenant jeter I (mais il faut absolument garder l'exécutable C).

C'est en gros ce qui s'est passé (mais avec Fortran.)

L'écriture d'un compilateur dans son propre langage est le *bootstrap* (*auto-amorçage*).

Poule et œuf du troisième millénaire

Soit un langage informatique $\mathcal{L}$ déjà compilable (et bootstrapé) sur une machine $\mathcal{M}$.

Je note $\mathcal{L} \overset{\mathcal{L}}{\to} \mathcal{M}$ le source et $\mathcal{L} \overset{\mathcal{M}}{\to} \mathcal{M}$ l'exécutable (du compilateur).

Une nouvelle machine $\mathcal{M}'$ arrive sur le marché.

- Changer le compilateur pour qu'il emette le code de $\mathcal{M}'$, soit un nouveau compilateur $\mathcal{L} \overset{\mathcal{L}}{\to} \mathcal{M}'$
- Le compiler, on obtient un *cross-compiler* $\mathcal{L} \overset{\mathcal{M}}{\to} \mathcal{M}'$.
- Compiler $\mathcal{L} \overset{\mathcal{L}}{\to} \mathcal{M}'$ avec le cross-compiler.
- Copier le résultat $\mathcal{L} \overset{\mathcal{M}'}{\to} \mathcal{M}'$ sur $\mathcal{M}'$.
- À tout hasard, sur $\mathcal{M}'$, compiler $\mathcal{L} \overset{\mathcal{L}}{\to} \mathcal{M}'$, avec $\mathcal{L} \overset{\mathcal{M}'}{\to} \mathcal{M}'$, normalement on a atteint un point fixe.

Si $\mathcal{L} \overset{\mathcal{L}}{\to} \mathcal{M}$ est buggé, il est parfois difficile de corriger.

Premiers pas

Une instruction :

- **Machine** Un (quelques) entier sur 32/64 bits, mais qui peut se représenter symboliquement (assembleur).

```
movl $n %eax
```

- **Machine abstraite** Un type Caml.

```
type ins = Ldi of int | ...
```

L'instruction (dite *load* ou *move* immédiat) range une constante entière dans un *registre*.

Un registre est une case de mémoire disponible à l'intérieur de l'unité de contrôle.

- **Machine :** un petit nombre de registres, ici `%eax`, `%ebx`...
- **Machine abstraite :** un seul registre dit accumulateur.

Opérations arithmétiques

Inteprétation de t_1 + t_2

```
let rec interp ... t = match t with
| Op (Add,t1,t2) ->
    let n1 = match interp ... t1 with Int n -> n in
    let n2 = match interp ... t2 with Int n -> n in
    Int (n1+n2)
...
```

Autrement dit,

- Interpréter `t1`, ranger le résultat dans `n1`,
- Interpréter `t2`, ranger le résultat dans `n2`,
- Rendre le résultat `n1+n2`.

Comment faire, avec la machine qui ne connaît, ni la liaison **let**, ni l'appel de fonction (et encore moins l'appel récursif).

Calculer avec une pile

Le *code* est une séquence d'instructions.

Remplacer « Interpréter `t`» par, executer le code qui calcule la valeur de `t` (dans l'accu), noté $\mathcal{C}(\texttt{t})$.

- Calculer `t1`, sauver l'accu dans la pile.
- Calculer `t2`.
- Additionner le contenu de l'accu et du sommet de pile (dans l'accu).

Soit compilation de t_1 `+` t_2.

$$\mathcal{C}(\texttt{t1}); \text{Push}; \mathcal{C}(\texttt{t2}); \text{Add}$$

NB : $\mathcal{C}$ est la fonction de compilation, des termes PCF vers le code.

Machine abstraite à pile

État de la machine : un triplet accu × pile × code, noté (A, S, C)

- Accumulateur : une valeur.
- Pile : suite de valeurs (en mémoire)
- Code : suite d'instruction (en mémoire)

Effet des instructions : sémantique à petit pas (système de transitions).

$$\begin{array}{rcl} (_, S, (\text{Ldi } n; C)) & \rightarrow & (n, S, C) \\ (A, S, (\text{Push}; C)) & \rightarrow & (A, (A; S), C) \\ (n_2, (n_1; S), (\text{Add}; C)) & \rightarrow & (n_1 + n_2, S, C) \end{array}$$

Le résultat du calcul est un état de la forme $(n, \emptyset, \emptyset)$.

Examples

Simple :

$$\mathcal{C}(\texttt{1+2}) = \text{Ldi } 1; \text{Push}; \text{Ldi } 2; \text{Add}$$

Simple :

$$\mathcal{C}(\texttt{3+4}) = \text{Ldi } 3; \text{Push}; \text{Ldi } 4; \text{Add}$$

Plus compliqué :

$$\begin{array}{rcl} \mathcal{C}(\texttt{(1+2)+(3+4)}) & = & \mathcal{C}(\texttt{1+2}); \text{Push}; \mathcal{C}(\texttt{3+4}); \text{Add} \\ & = & \left\{ \begin{array}{l} \text{Ldi } 1; \text{Push}; \text{Ldi } 2; \text{Add}; \\ \text{Push}; \\ \text{Ldi } 3; \text{Push}; \text{Ldi } 4; \text{Add}; \\ \text{Add} \end{array} \right. \end{array}$$

À la fin, l'accumulateur contient 10 et la pile est comme au début.

Machine abstraite en Caml

- Types de données.

```
let value = Int of int | ...
and stack = value list

type ins = Ldi of integer | Push | Cop of op | ...
and code = ins list (* code = liste d'instructions *)
```

- Exécution de la machine.

```
let rec run a s c = match a,s,c with
| _,_,(Ldi i::c) -> run (Int i) s c
| _,_,(Push::c)  -> run a (a::s) c
| Int n2,(Int n1::s),(Cop Add::c) ->
    run (Int (n1+n2)) s c
...
| v,[],[] -> v (* résultat final *)
```

La pile en vrai

Il est assez facile de réaliser une *pile* en machine.

- La mémoire est un grand tableau d'entiers, les indices sont appelés *addresses*.
- La pile est une zone de la mémoire.
- Empiler (*push*) le contenu de l'accumulateur :

```
pushl %eax
```

- Dépiler (*pop*) le sommet de la pile dans l'accumulateur.

```
popl %eax
```

Si la machine donne directement les instructions *push* et *pop*.

Sans instructions *push*/*pop*

L'addresse du sommet de la pile est dans un registre particulier dit pointeur de pile (`%esp`).

Push

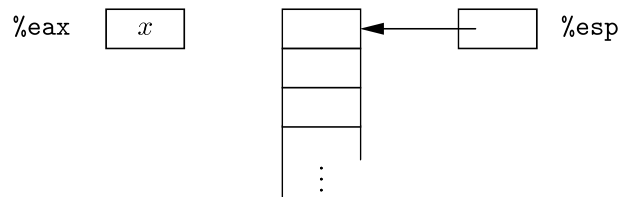
```
subl $4,%esp
movl %eax,0(%esp)
```

Pop

```
movl 0(%esp),%eax
addl $4,%eax
```

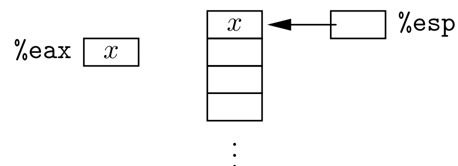

Push sur la vraie machine

État de la vraie machine :

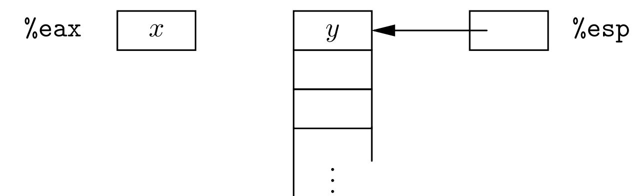


- **Push :**

```
subl $4,%esp
movl %eax,0(%esp)
```

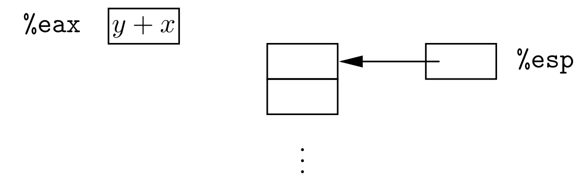


Addition en pile sur la vraie machine



- **Add** :

```
addl %eax,0(%esp)
popl %eax
```



Compilation

Comme une définition de la fonction $\mathcal{C}$.

$$\begin{aligned}\mathcal{C}(\mathtt{n}) &= \mathrm{Ldi}\ n \\ \mathcal{C}(t_1\ \ op\ \ t_2) &= \mathcal{C}(t_1); \mathrm{Push}; \mathcal{C}(t_2); \mathrm{Cop}\ \ op\end{aligned}$$

En Caml.

```
let rec compile t = match t with
| Num n -> [Ldi n]
| Op (op,t1,t2) -> compile t1 @ [Push] @ compile t2 @ [Cop op]
| ...
```

Ou encore (éviter les concaténations).

```
let rec compile_k t k = match t with
| Num n -> Ldi i::k
| Op (op,t1,t2) -> compile_k t1 (Push::compile_k t2 (Cop op::k))
```

(`compile` t `@` c = `compile_k` t c, pour tout terme t et tout code c)

La conditionelle

Interprétation.

$$\frac{E \vdash t_1 \hookrightarrow \mathtt{0} \qquad E \vdash t_2 \hookrightarrow v}{E \vdash \textbf{Ifz}\ t_1\ \textbf{Then}\ t_2\ \textbf{Else}\ t_3 \hookrightarrow v}$$

$$\frac{E \vdash t_1 \hookrightarrow n (n \neq \mathtt{0}) \qquad E \vdash t_2 \hookrightarrow v}{E \vdash \textbf{Ifz}\ t_1\ \textbf{Then}\ t_2\ \textbf{Else}\ t_3 \hookrightarrow v}$$

La compilation semble évidente :

- Exécuter $\mathcal{C}(t_1)$ (le résultat est un entier dans l'accu).
- ▷ Si l'accu contient 0 alors exécuter $\mathcal{C}(t_2)$.
  ▷ Si l'accu contient $n \neq 0$, alors exécuter $\mathcal{C}(t_3)$.
  ▷ Sinon, c'est une erreur (libre à nous de la détecter ou pas).

Tout simplement

On se donne une instruction $\mathrm{Test}(C_2, C_3)$ qui va exécuter C_2 ou C_3 selon la valeur de l'accumulateur. Genre

$$\begin{aligned}(0, S, \mathrm{Test}(C_2, C_3)) &\rightarrow (0, S, C_2) \\ (n, S, \mathrm{Test}(C_2, C_3)) &\rightarrow (n, S, C_3) \quad (n \neq 0)\end{aligned}$$

Et on compile :

$$\mathcal{C}(\textbf{Ifz}\ t_1\ \textbf{Then}\ t_2\ \textbf{Else}\ t_3) = \mathcal{C}(t_1); \mathrm{Test}(\mathcal{C}(t_2), \mathcal{C}(t_3))$$

Pas si simple

Considérer :

(**Ifz** t_1 **Then** t_2 **Else** t_3)+3

On aura un code du genre

$$\mathcal{C}(t_1); \mathrm{Test}(\mathcal{C}(t_2), \mathcal{C}(t_3)); \mathrm{Push}; \mathrm{Ldi}\ 3; \mathrm{Add}$$

C'est à dire que les transitions de la machine abstraite sont plutôt :

$$\begin{aligned}(0, S, (\mathrm{Test}(C_2, C_3); C) &\rightarrow (0, S, (C_2; C)) \\ (n, S, (\mathrm{Test}(C_2, C_3); C)) &\rightarrow (n, S, (C_3; C)) \quad (n \neq 0)\end{aligned}$$

Dans une implémentation Caml, on écrira malheureusement:

```
let rec run a s c = match a,s,c with
...
| Int 0,_,(Test (c2,c3)::c) -> run a s (c2@c)
```

Mais on peut éviter la concaténation avec un peu de reflexion.

Conditionelle, vraie machine

Dans la vraie machine, pas de liste d'instruction en argument des instructions !

Une seule séquence, et des instructions de saut.

```
    C(t1)
    testl %eax,%eax
    je L2          #Test (L2,L3)
L3:
    C(t3)
    jmp Lk
L2:
    C(t2)
Lk:
    pushl %eax      #Push
    movl $3,%eax    #Ldi 3
    addl 0(%esp)
    addl $4,%esp    #Add
```


Les environnements

Les règle de la variable et du **Let**.

$$\frac{E(x) = v}{E \vdash x \hookrightarrow v} \qquad \frac{E \vdash t_1 \hookrightarrow v_1 \qquad E \oplus [x = v_1] \vdash t_2 \hookrightarrow v}{E \vdash \textbf{Let } x = t_1 \textbf{ In } t_2 \hookrightarrow v}$$

(Tiens nous sommes en appel par valeur).

On ajoute donc:

- Un composant « environnement » à la machine dont l'état devient (A, S, E, C).
- Une instruction Extend pour ajouter une liaison.
- Une instruction Search pour trouver la valeur d'une variable.

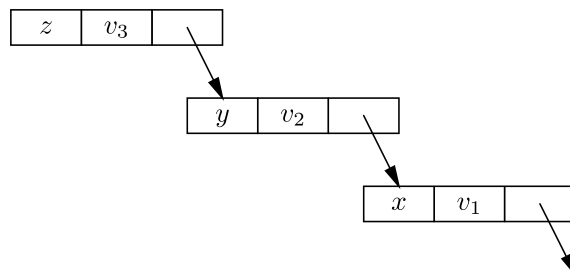
Si Search appelle **List**.`assoc`, on a pas compilé grand chose.

Retour sur les accès aux variables

Considérons :

Let x = t_1 **In**
Let y = t_2 **In**
Let z = t_3 **In**
t

Au moment d'interpréter t, l'environnement est de de la forme :



Compilation des accès aux variables

Au vu du programme la position des variables dans l'environnement à l'exécution (E) est connue (pour z, c'est 0,..., pour x c'est 2).

On peut donc simplifier E: c'est une liste de valeurs.

$$\begin{array}{rcl} (A, S, E, (\text{Extend}; C)) & \rightarrow & (A, S, (A; E), C) \\ (_, S, (v_0; v_1; \cdots; v_k; \cdots), (\text{Search } k; C)) & \rightarrow & (v_k, S, (v_0; \cdots; v_k; \cdots), C) \end{array}$$

Compilation, $\mathcal{C}_E(t)$ où E donne les positions des variables (liste de variables suffit ici).

$$\begin{array}{rcl} \mathcal{C}_E(\textbf{Let } x = t_1 \textbf{ In } t_2) & = & \mathcal{C}_E(t_1); \text{Extend}; \mathcal{C}_{x;E}(t_2) \\ \mathcal{C}_{[x_0;\cdots;x_k;\cdots]}(x_k) & = & \text{Search k} \end{array}$$

Mais attention

```
Let x = 2 In
(Let x = 1 in x+x)+x
```

Il faut après le code $\mathcal{C}_{[x]}$(**Let** `x = 1` **In** $x + x$) redonner sa valeur d'origine à l'environnement.

Une solution simple est de sauver/restaurer l'environnement à l'aide de la pile S.

$$\begin{array}{rcl} (A, S, E, (\text{PushEnv}; C)) & \rightarrow & (A, (E; S), E, C) \\ (A, (E; S), _, (\text{PopEnv}; C)) & \rightarrow & (A, S, E, C) \end{array}$$

Et compilation complète du **Let**.

$$\mathcal{C}_E(\textbf{Let } x \texttt{ = } t_1 \textbf{ In } t_2) \quad = \quad \text{PushEnv}; \mathcal{C}_E(t_1); \text{Extend}; \mathcal{C}_{x;E}(t_2); \text{PopEnv}$$

La vraie machine

Différence essentielle entre pile et environnement :

- Les cases de pile sont allouées (Push) et désallouées (Pop) explicitement. Une compilation correcte garantit que la pile est rendue comme on l'a trouvée.
- Pour l'environnement, il n'est pas possible de désallouer explicitement en général (à cause des fermetures).

Il fait donc ajouter du *code supplémentaire* au code compilé, qui gère l'allocation mémoire (facile) et la récupération (plus dur). Ce code supplémentaire s'appelle parfois un *environnement d'exécution* (argl) ou encore le support à l'exécution.

Mais le principe de changer les noms de variables en positions est typique.

Machine abstraite en Caml

On utilise la gestion mémoire de Caml.

```
let rec run a s e c = match a,s,c with
...
| _,_,(Extend::c) -> run a s (a::e) c
...
```

On utilise donc le support à l'exécution de Caml, sans se poser plus de questions.

Mais si on écrit une machine abstraite en assembleur (en C), il faudra se poser des questions (réponse `malloc` + écrire un GC).

Fonctions

Interprétation de la fonction.

$$E \vdash \textbf{Fun } x \texttt{ -> } t \hookrightarrow \langle x \bullet t \bullet E \rangle$$

La fermeture en machine est une paire $\langle C \bullet E \rangle$ (t compilé, et plus besoin du nom de la variable).

Par exemple :

```
Let y = t_y In
Let f = Fun x -> x+y In
...
```

La fermeture suivante représente f.

$$\langle \text{Search } 0; \text{Push}; \text{Search } 1; \text{Add} \bullet v_y \rangle$$

Compilation

Une nouvelle instruction pour fabriquer les fermetures :

$$(_, S, E, (\text{MakeClo } [C]; C')) \quad \rightarrow \quad (\langle C \bullet E \rangle, S, E, C')$$

On ose à peine préciser :

```
let rec run a s e c  = match a,s,c with
...
| _,_,(MakeClo c::c') -> run (Clo (c,e)) s e c'
...
```

Compilation, seuls les environnements sont (un peu) subtils.

$$\mathcal{C}_E(\textbf{Fun } x \texttt{ -> } t) \quad = \quad \text{MakeClo } [\mathcal{C}_{x;E}(t)]$$

Noter quand même qu'une fermeture est la valeur d'une fonction, et se retrouve donc logiquement dans l'accumulateur.

Application

La règle d'intepprétation.

$$\frac{E \vdash t_1 \hookrightarrow \langle x \bullet t \bullet E' \rangle \qquad E \vdash t_2 \hookrightarrow v_2 \qquad E' \oplus [x = v_2] \vdash t \hookrightarrow v}{E \vdash t_1 \; t_2 \hookrightarrow v}$$

Soit il faut :

- Changer l'environnement $\Rightarrow E' \oplus [x = v_2]$
- Interpréter t_2 dans ce nouvel environnement.

Donc une instruction Apply fait tout ça

$$(\langle C' \bullet E' \rangle, (v; S), E, \text{Apply}) \quad \rightarrow \quad (A, \emptyset, (v; E'), C')$$

Et compilation :

$$\mathcal{C}_E(t_1 \; t_2) \quad = \quad \mathcal{C}_E(t_2); \text{Push}; \mathcal{C}_E(t_1); \text{Apply}$$

Mais une fois de plus c'est un peu trop simple. . .

Application en contexte

```
Let x = 1 In
Let f = Fun x -> x+x In
(f 2) + x
```

On a donc, par compilation, un code du genre:

$$\text{Ldi } 2; \text{Push}; \text{Search } 0; \text{Apply}; \mathcal{C}_E(\texttt{x}); \text{Add}$$

(ou E est $[\texttt{f}; \texttt{x}; \cdots]$)

Il faut donc retrouver l'état de la machine après le retour de la fonction.

Sauver l'état de la machine, sur la pile

Un état de machine est normalement (A, S, E, C), mais

- L'appel de fonction rend son résultat dans A, inutile de sauver A avant l'appel.
- Si on sauve l'état sur la pile, il est bizarre de sauver la pile.
- On ne sauve donc que E et C (code après apply), ce sera le travail de Apply.

$$(\langle C' \bullet E' \rangle, (v; S), E, (\text{Apply}; C)) \quad \rightarrow \quad (A, (C; E; S), (v; E'), C')$$

- Et une règle spéciale pour la fin du code d'une fonction.

$$(A, (C; E; S), _, \emptyset) \quad \rightarrow \quad (A, S, E, C)$$

Compilation de l'appel de fonction

Avec Apply qui sauve l'état de la machine, c'est simple.

$$\mathcal{C}_E(t_1\ t_2) \quad = \quad \mathcal{C}_E(t_2); \text{Push}; \mathcal{C}_E(t_1); \text{Apply}$$

Compte tenu de ce qui vient d'être vu, à la fin de la séquence ci dessus.

- La pile S a retrouvé sa valeur du début.
- L'environnement (à l'exécution) a retrouvé sa valeur du début
- Et A contient la valeur rendue par la fonction.

Modèle du poly

Sauvegarde et restauration explicite de l'environnement par l'appelant.

$$\mathcal{C}_E(t_1\ t_2) = \text{PushEnv}; \mathcal{C}_E(t_2); \text{Push}; \mathcal{C}_E(t_1); \text{Apply}; \text{PopEnv}$$

Avec un Apply moins puissant :

$$(A, (E; S), _, (\text{PopEnv}; C)) \quad \rightarrow \quad (A, S, E, C)$$
$$(\langle C' \bullet E' \rangle, (v; S), E, (\text{Apply}; C)) \quad \rightarrow \quad (A, S, (v; E'), (C'; C))$$

N.B. La compilation du poly, produit toujours Apply suivi de PopEnv.

Le modèle du poly n'a pas de règle spéciale pour le retour de fonction (mais il triche un peu dans la règle de Apply).

Un miracle

Soit une fonction à deux arguments:

```
Let k = Fun x -> Fun y -> x+y In
k 1 2
```

Soit **Let** `k` = t_k **In** t. Le code de t_k est :

$$\text{MakeClo [MakeClo [Search 1; Push; Search 0; Add]]}$$

Son exécution construit la fermeture $c = \langle \text{MakeClo } C \bullet \emptyset \rangle$, avec $C = [\text{Search } 1; \cdots]$.

Et le code de t est :

Ldi 2; Push;

Ldi 1; Push; Search 0; Apply;

Apply;

Appel de fonction à deux arguments

Au moment du premier Apply.

$$(\langle \text{MakeClo } [C] \bullet \emptyset \rangle, (1; 2), E, (\text{Apply}; \text{Apply}; \emptyset))$$

(Où E est l'environnement donne sa valeur à `k`.)

Transition Apply

$$(_, ((\text{Apply}; \emptyset); E; 2), (1; E), \text{MakeClo } C)$$

Exécution de MakeClo C.

$$(\langle C \bullet (1; E) \rangle, ((\text{Apply}; \emptyset); E; 2), (1; E), \emptyset)$$

Retour de fonction.

$$(\langle C \bullet (1; E) \rangle, (2), E, (\text{Apply}; \emptyset))$$

Nouvel Apply :

$$(_, ((\emptyset); E), (2; 1; E), C)$$

C'est à dire que le code $C = [\text{Search } 1; \text{Push}; \text{Search } 0; \text{Add}]$ s'exécute dans l'environnement approprié ($\mathtt{x} = 1$ en seconde position, $\mathtt{y} = 2$ en première).

À la fin du code C.

$$(3, ((\emptyset); E), (2; 1; E), \emptyset)$$

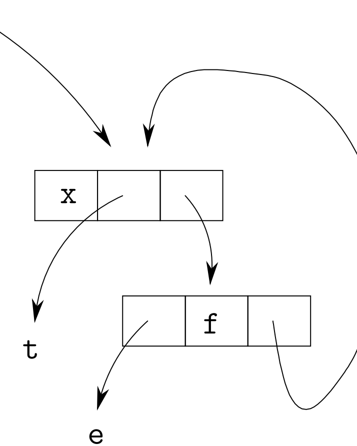
Et retour de fonction.

$$(3, \emptyset, E, \emptyset)$$

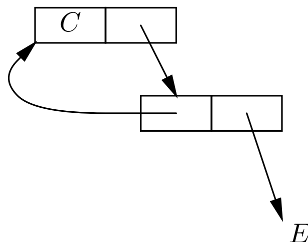
Le point fixe

Nous ne considérons que le point fixe de fonctions **Fix** f `->` **Fun** x `->` t.

Interprétation par une fermeture bouclée.



On invente une nouvelle instruction MakeCloRec qui construit ce style de fermeture bouclée, mais pour la machine.



Avec donc la règle d'exécution :

$$(_, S, E, (\text{MakeCloRec } [C]; C')) \rightarrow (c, S, E, C'), \quad \text{où } c \text{ graphe } c = \langle C \bullet c; E \rangle$$

À comparer avec :

$$(_, S, E, (\text{MakeClo } [C]; C')) \rightarrow (\langle C \bullet E \rangle, S, E, C')$$

On ose à peine préciser

Exécution en Caml.

```
let rec run a s e c  = match a,s,c with
...
| _,_,(MakeCloRec c::c') ->
  let rec clo_rec = Clo (c,clo_rec::e) in
  run clo_rec s e c'
```

Compilation, seuls les environnements sont (un peu) subtils.

$$\mathcal{C}_E(\textbf{Fix } f \texttt{ -> } \textbf{Fun } x \texttt{ -> } t) \quad = \quad \text{MakeCloRec } [\mathcal{C}_{x;f;E}(t)]$$

N.B. La transition Apply ne change pas.

Évidemment rien n'empèche de se passer de MakeClo.

$$\mathcal{C}_E(\textbf{Fun } x \texttt{ -> } t) \quad = \quad \text{MakeCloRec } [\mathcal{C}_{x;_;E}(t)]$$

- ▶ TP, Compilateur et machine abstraite (en Caml) à deux.
- ▶ La prochaine fois, PCF typé.

Les transitions de la machine (A, S, E, C)

$$
\begin{array}{rcll}
(_, S, E, (\text{Ldi } n; C)) & \to & (n, S, E, C) & \\
(A, S, E, (\text{Push}; C)) & \to & (A, (A; S), E, C) & \\
(n_2, (n_1; S), E, (\text{Add}; C)) & \to & (n_1 + n_2, S, E, C) & \\
(0, S, E, (\text{Test}(C_2, C_3); C)) & \to & (0, S, E, (C_2; C)) & \\
(n, S, E, (\text{Test}(C_2, C_3); C)) & \to & (n, S, E, (C_3; C)) & (n \neq 0) \\
(A, S, E, (\text{Extend}; C)) & \to & (A, S, (A; E), C) & \\
(_, S, (v_0; v_1; \cdots; v_k; \cdots), (\text{Search } k; C)) & \to & (v_k, S, (v_0; \cdots; v_k; \cdots), C) & \\
(A, S, E, (\text{PushEnv}; C)) & \to & (A, (E; S), E, C) & \\
(A, (E; S), _, (\text{PopEnv}; C)) & \to & (A, S, E, C) &
\end{array}
$$

Transitions pour les fonctions

$$
\begin{array}{rcl}
(_, S, E, (\text{MakeClo } [C]; C')) & \to & (\langle C \bullet E \rangle, S, E, C') \\
(_, S, E, (\text{MakeCloRec } [C]; C')) & \to & (c, S, E, C'), \quad \text{avec } c = \langle C \bullet c; E \rangle \\
(\langle C' \bullet E' \rangle, (v; S), E, (\text{Apply}; C)) & \to & (A, (C; E; S), (v; E'), C') \\
(A, (C; E; S), _, \emptyset) & \to & (A, S, E, C)
\end{array}
$$

État final de la machine :

$$(v, \emptyset, _, \emptyset)$$

(Pile et code vides). Le résultat est v, le contenu de l'accumulateur.

Compilation de PCF

$$
\begin{array}{rcl}
\mathcal{C}_E(\texttt{n}) & = & \text{Ldi } n \\
\mathcal{C}_E(t_1 \texttt{ + } t_2) & = & \mathcal{C}_E(t_1); \text{Push}; \mathcal{C}_E(t_2); \text{Add} \\
\mathcal{C}_E(\textbf{Ifz } t_1 \textbf{ Then } t_2 \textbf{ Else } t_3) & = & \mathcal{C}_E(t_1); \text{Test}(\mathcal{C}_E(t_2), \mathcal{C}_E(t_3)) \\
\mathcal{C}_E(\textbf{Let } x \texttt{ = } t_1 \textbf{ In } t_2) & = & \text{PushEnv}; \mathcal{C}_E(t_1); \text{Extend}; \mathcal{C}_{x;E}(t_2); \text{PopEnv} \\
\mathcal{C}_{[x_0; \cdots; x_k; \cdots]}(x_k) & = & \text{Search k} \\
\mathcal{C}_E(t_1 \ t_2) & = & \mathcal{C}_E(t_2); \text{Push}; \mathcal{C}_E(t_1); \text{Apply} \\
\mathcal{C}_E(\textbf{Fun } x \texttt{ -> } t) & = & \text{MakeClo } [\mathcal{C}_{x;E}(t)] \\
\mathcal{C}_E(\textbf{Fix } f \texttt{ -> } \textbf{Fun } x \texttt{ -> } t) & = & \text{MakeCloRec } [\mathcal{C}_{x;f;E}(t)]
\end{array}
$$

Sauf erreur, bien entendu.