Compilation

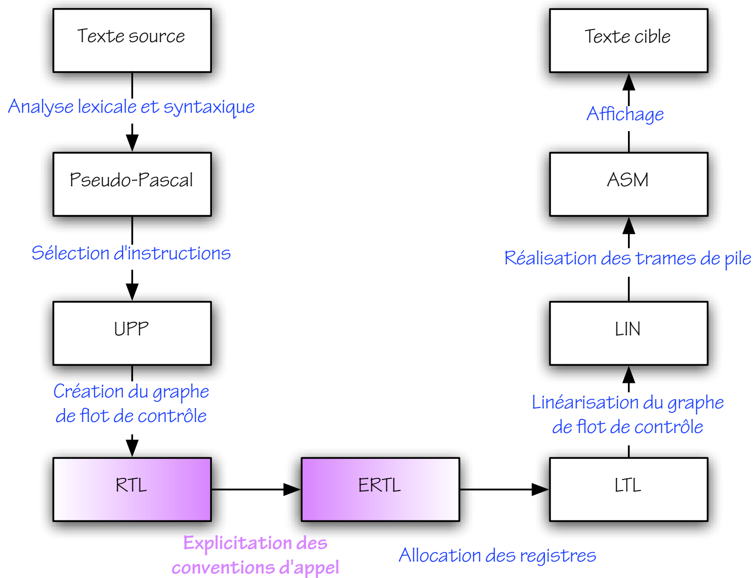
Cours n°5: Explicitation de la convention d'appel: de RTL à ERTL

Sandrine Blazy
(d'après le cours de François Pottier)

ensiie - 2e année

24 novembre 2008

école nationale supérieure d'informatique
pour l'industrie et l'entreprise



ensiie

La convention d'appel

La *convention d'appel* est le résultat d'un contrat entre plusieurs parties :

- l'appelant (« caller ») ;
- l'appelé (« callee ») ;
- le système d'exécution (« runtime system ») ;
- le système d'exploitation (« operating system »).

Elle permet d'abord la *communication* entre appelant et appelé. Elle peut dans certains cas être exploitée par le système d'exécution (par exemple, par le glaneur de cellules) ou par le système d'exploitation (par exemple, par un gestionnaire d'interruptions).

Quand expliciter la convention d'appel ?

La convention d'appel est toujours *invisible* dans le langage source, car celui-ci est « de haut niveau ».

L'expliciter fait apparaître de *nouvelles notions :* registres physiques, trames de pile, emplacements de pile. Cela complique le langage intermédiaire, donc autant le faire aussi tard que possible.

Néanmoins, l'allocation de registres *dépend* de la convention d'appel.

Nous explicitons donc la convention d'appel *immédiatement avant* la phase d'allocation de registres.

1. De RTL à ERTL

Remarques

Optimisation des appels terminaux

Fonctions imbriquées et fonctions de première classe

Explicit Register Transfer Language (ERTL)

Dans ERTL, la *convention d'appel* est explicitée.

- *paramètres* et, le cas échéant, *résultat* des procédures et fonctions sont transmis à travers des *registres physiques* et/ou des *emplacements de pile* ;
- procédures et fonctions ne sont plus distinguées ;
- *l'adresse de retour* devient un paramètre explicite ;
- l'allocation et la désallocation des *trames de pile* devient explicite ;
- les registres physiques *callee-save* sont *sauvegardés* de façon explicite.

Explicit Register Transfer Language (ERTL)

Voici une traduction de la fonction factorielle dans ERTL :

```
procedure f(1)
var %0, %1, %2, %3, %4, %5, %6
entry f11
f11: newframe              → f10
f10: move    %6, $ra       → f9
f9 : move    %5, $s1       → f8
f8 : move    %4, $s0       → f7
f7 : move    %0, $a0       → f6
f6 : li      %1, 0         → f5
f5 : blez    %0            → f4, f3
f3 : addiu   %3, %0, −1    → f2
f2 : j                     → f20
f20: move    $a0, %3       → f19
f19: call    f(1)          → f18
f18: move    %2, $v0       → f1
f1 : mul     %1, %0, %2    → f0
f0 : j                     → f17
f17: move    $v0, %1       → f16
f16: move    $ra, %6       → f15
f15: move    $s1, %5       → f14
f14: move    $s0, %4       → f13
f13: delframe              → f12
f12: jr      $ra
f4 : li      %1, 1         → f0
```

Les registres physiques du MIPS

Le module MIPS définit un type abstrait pour représenter les registres physiques du processeur :

```
type register

val equal : register → register → bool
val print : register → string

module RegisterSet : ...
module RegisterMap : ...
```

ensiie

La convention d'appel du MIPS

Le module MIPS indique quels registres physiques sont utilisés pour passer des paramètres, passer l'adresse de retour, et renvoyer un résultat :

```
val parameters: register list
val ra: register
val result: register
```

Les paramètres excédentaires sont passés sur la pile.

La convention d'appel du MIPS

Le module MIPS indique également quels registres doivent être préservés par l'appelé lors d'un appel de procédure :

```
val callee_saved: RegisterSet.t
```

ensiie

Nouvelles instructions en ERTL (I)

Le jeu d'instructions de ERTL permet l'accès, en lecture et en écriture, aux *registres physiques :*

```
type instruction =
  | ...
  | IGetHwReg of Register.t * MIPS.register * Label.t
  | ISetHwReg of MIPS.register * Register.t * Label.t
  | ...
```

Nouvelles instructions en ERTL (II)

ERTL permet également l'accès à certains *emplacements de pile :*

```
type instruction =
  | ...
  | IGetStack of Register.t * slot * Label.t
  | ISetStack of slot * Register.t * Label.t
  | ...
```

Mais *qu'est-ce* qu'un emplacement de pile ? Et *comment déterminer* dès maintenant à quel emplacement on souhaitera placer telle ou telle donnée ?

Régions de pile

Chaque procédure utilise la pile pour *accéder* aux paramètres qu'elle a reçus et pour *transmettre* des paramètres aux procédures qu'elle-même appelle.

Du point de vue de chaque procédure, on distingue donc *deux régions* de la pile : celle des paramètres *entrants* et celle des paramètres *sortants*.

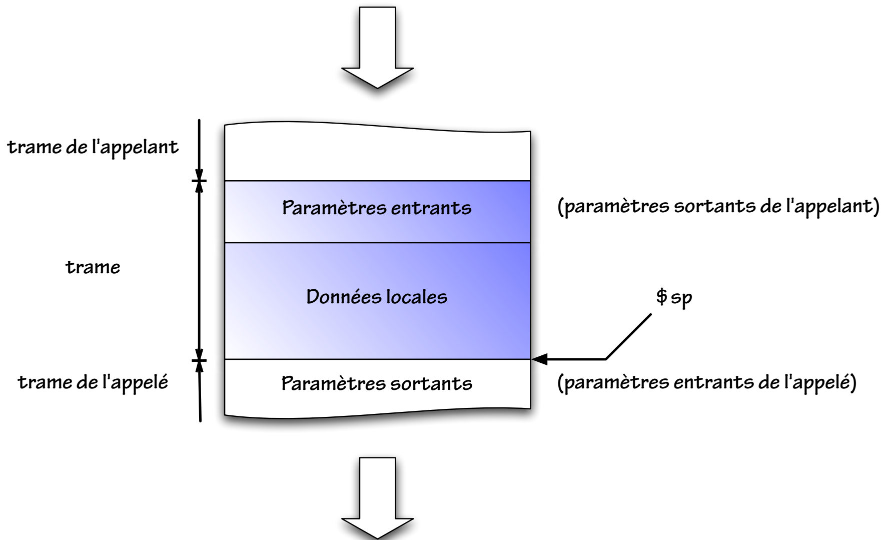
Après l'allocation de registres, chaque procédure aura besoin d'une troisième région pour contenir des données *locales*.

Trames de pile

Chaque procédure a *sa propre vision* de la pile. On appelle *trame de pile* (« stack frame ») la portion de pile allouée pour chaque procédure.

Chaque trame est sous-divisée en régions. Lors d'un appel de procédure, la région des paramètres *sortants* de l'appelant *coïncide* avec celle des paramètres *entrants* de l'appelé.

C'est ainsi que *communiquent* appelant et appelé. Ceci exige bien sûr de s'accorder sur l'emplacement exact de chaque paramètre.


trame de l'appelant
Paramètres entrants
(paramètres sortants de l'appelant)
trame
Données locales
$ sp
trame de l'appelé
Paramètres sortants
(paramètres entrants de l'appelé)

Régions et emplacements

Un emplacement de pile est identifié par la *région* dans laquelle il se trouve et par un *décalage* au sein de cette région. Dans le cas de ERTL, on a deux régions :

```
type slot =
  | SlotIncoming of offset
  | SlotOutgoing of offset
```

Les décalages commencent à 0 et augmentent de 4 en 4 à chaque fois qu'un nouvel emplacement est alloué.

Ces emplacements seront traduits en décalages vis-à-vis de $\$sp$ lors d'une phase ultérieure, une fois la taille de chaque région connue.

ensiie

Nouvelles instructions en ERTL (III)

Concrètement, chaque procédure accèdera au contenu de sa trame de pile à l'aide du registre $\$sp$, qui sera *décrémenté* au début de la procédure et *restauré* à la fin. Mais de combien ? En ERTL, on ne le sait pas encore, donc on utilise deux instructions spéciales :

```
type instruction =
  | ...
  | INewFrame of Label.t
  | IDeleteFrame of Label.t
  | ...
```

L'accès aux pseudo-registres et aux emplacements de pile n'est permis qu'*après* avoir exécuté INewFrame et *avant* d'exécuter IDeleteFrame. (Pourquoi ?)

ensiie

Nouvelles instructions en ERTL (IV)

Dans ERTL, l'instruction *ICall* ne mentionne pas les arguments, qui doivent avoir été *préalablement rangés* aux endroits convenus. Symétriquement, le résultat doit être *lu a posteriori* à l'endroit convenu. Procédures et fonctions ne sont plus distinguées.

```
type instruction =
  | ...
  | ICall of Primitive.callee * int32 * Label.t
  | IReturn
  | ...
```

ICall stocke l'adresse de retour dans le registre physique $\$ra$. Symétriquement, une nouvelle instruction *IReturn* effectue un saut vers l'adresse stockée dans $\$ra$.

ensiie

De RTL à ERTL, en résumé

L'explicitation de la convention d'appel apporte au code des modifications localisées :

- *autour de chaque appel* de procédure ;
- en *début* et *fin* de procédure.

En résumé,

- des **move** entre pseudo-registres et registres physiques ou emplacements de pile apparaissent pour *envoyer ou recevoir paramètres ou résultat* et pour *sauvegarder les registres « callee-save »* ;
- **newframe** et **delframe** apparaissent en début et fin de procédure.

ensiie

De RTL à ERTL, en résumé

Revoici la traduction de la fonction factorielle dans ERTL :

```
procedure f(1)
var %0, %1, %2, %3, %4, %5, %6
entry f11
f11: newframe            → f10
f10: move   %6, $ra      → f9
f9 : move   %5, $s1      → f8
f8 : move   %4, $s0      → f7
f7 : move   %0, $a0      → f6
f6 : li     %1, 0        → f5
f5 : blez   %0           → f4, f3
f3 : addiu  %3, %0, −1   → f2
f2 : j                   → f20

f20: move   $a0, %3      → f19
f19: call   f(1)         → f18
f18: move   %2, $v0      → f1
f1 : mul    %1, %0, %2   → f0
f0 : j                   → f17
f17: move   $v0, %1      → f16
f16: move   $ra, %6      → f15
f15: move   $s1, %5      → f14
f14: move   $s0, %4      → f13
f13: delframe            → f12
f12: jr     $ra
f4 : li     %1, 1        → f0
```

De RTL à ERTL

2 Remarques

Optimisation des appels terminaux

Fonctions imbriquées et fonctions de première classe

ensiie

À propos du passage de paramètres sur la pile

Le passage de paramètres sur la pile était utilisé de façon *uniforme*, pour tous les paramètres, jusque dans les années 1970.

Depuis, on préfère spécifier que *les k premiers paramètres sont passés dans des registres* – typiquement, $k = 4$ ou $k = 6$. L'objectif est de gagner du temps en limitant le trafic mémoire.

Mais, si une procédure a *reçu* un paramètre dans $\$a0$, alors, lorsqu'elle souhaite appeler elle-même une autre procédure, elle doit également utiliser $\$a0$ pour lui *envoyer* un argument, donc doit typiquement d'abord *sauvegarder* le contenu de $\$a0$ sur la pile.

En quoi a-t-on gagné du temps ?

ensiie

À propos du passage de paramètres sur la pile

Plusieurs réponses sont possibles :

- peut-être la procédure qui s'apprête à effectuer l'appel n'aura-t-elle *plus besoin* de l'ancienne valeur de $\$a0$, qui ne sera alors pas sauvegardée ;
- une *procédure feuille,* qui n'appelle aucune autre procédure, n'aura typiquement pas besoin de sauvegarder en mémoire les paramètres qu'elle a reçus ;
- certains compilateurs effectuent une *allocation de registres interprocédurale*, qui permet de choisir une convention d'appel distincte pour chaque procédure.

ensiie

Vers des trames de pile plus complexes

Notre petit compilateur utilise deux régions de la trame de pile (*entrants* et *sortants*) pour stocker les paramètres entrants et sortants, plus une troisième (*locaux*) pour les pseudo-registres « spillés ».

Un compilateur plus complexe utiliserait des régions *plus nombreuses*, selon un schéma qui pourrait *dépendre* de l'architecture cible (processeur, système d'exploitation, système d'exécution).

Consultez par exemple « Declarative composition of stack frames », de Lindig et Ramsey, dont l'approche est élégante.

Vers des conventions d'appel plus complexes

Pseudo-Pascal ne permet de manipuler que des données *entières de la taille d'un mot*, ce qui simplifie beaucoup la convention d'appel.

Dans un compilateur réel, on doit gérer des entiers de *diverses tailles*, des nombres *réels* en virgule flottante, ainsi que les *irrégularités* du ou des processeurs cible.

Consultez par exemple « Staged allocation : ... », d'Olinsky, Lindig et Ramsey, pour avoir une idée de la difficulté.

ensiie

De RTL à ERTL

Remarques

3 Optimisation des appels terminaux

Fonctions imbriquées et fonctions de première classe

ensiie

De la récursivité comme forme d'itération

La version récursive de la fonction factorielle :

```
function f (n : integer) : integer;
begin
   if n<= 0 then
      f := 1
   else
      f := n * f (n − 1)
end;
```

donne lieu à du code beaucoup moins efficace que la version itérative basée sur une boucle **while**. (Pourquoi ?)

ensiie

De la récursivité comme forme d'itération

Qu'en est-il de cette version, modifiée pour utiliser un *accumulateur*, où la multiplication est effectuée *avant* l'appel récursif ?

```
function f (n, accu : integer) : integer;
begin
   if n <= 1 then
      f := accu
   else
      f := f (n - 1, n * accu)
end;
```

L'appel à f est la *dernière* opération effectuée par la fonction, et le résultat de cet appel est *transmis :* il devient le résultat de la fonction. Un appel qui réunit ces deux conditions est dit *terminal* (« tail call »).

ensiie

Optimisation des appels terminaux

Supposons que f effectue un appel terminal vers g. Alors, dès que g rendra la main à f, celle-ci s'empressera de *détruire* sa trame de pile et de *rendre la main* à son propre appelant.

Dans ce cas, la trame de pile associée à f n'a plus de raison d'être. On pourrait la détruire *avant* l'appel à g et faire en sorte que g rende la main *directement* à l'appelant de f.

Optimisation des appels terminaux

Un appel terminal de f à g est donc compilé comme suit :

- placer les arguments attendus par g dans les registres physiques dédiés, comme pour un appel normal ;
- *restaurer* la valeur initiale des registres « callee-save », *y compris* $\$ra$ *;*
- *désallouer* la trame de pile de f ;
- transférer le contrôle, par un *simple saut* – **j** et non **jal** – à g.

Du point de vue de g, tout se passe comme s'il était appelé non pas par f mais par l'appelant de f.

ensiie

Retour sur la factorielle

Dans le cas de la factorielle avec accumulateur, le *seul* registre sauvegardé sur la pile, avant optimisation, est $\$ra$. Il doit être sauvegardé car sa valeur est utilisée *après* l'appel récursif à f, lequel écrase $\$ra$.

Or, l'élimination des appels terminaux *déplace* l'utilisation de $\$ra$ et la situe *avant* le saut final vers f. Donc *il n'est plus nécessaire de sauvegarder $\$ra$,* ce que l'allocation de registres déterminera sans difficulté.

Donc la trame de pile associée à f devient *de taille nulle* et *disparaît*.

Retour sur la factorielle

Enfin, l'allocateur de registres n'aura aucun mal à placer le résultat des expressions n $-$ 1 et n $*$ accu *directement dans les registres conventionnels* $\$a0$ et $\$a1$, évitant ainsi deux instructions **move**.

De ce fait, l'appel terminal à f est traduit par *un simple saut* **j**.

Ce qui signifie que la version récursive avec accumulateur et la version itérative de la fonction factorielle conduisent à un code *identique*.

ensiie

Un cas particulier

Lorsqu'une fonction f effectue un appel terminal à elle-même, il est inutile de *désallouer* la trame de f et de *restaurer* les registres « callee-save » pour aussitôt *réallouer* une trame et *sauvegarder* à nouveau les registres.

Dans ce cas, l'appel terminal peut être traduit comme suit :

- placer les arguments attendus par f dans les registres physiques dédiés, comme pour un appel normal ;
- transférer le contrôle, par un simple saut **j**, au point situé dans f *après* l'allocation de la trame et la sauvegarde des registres « callee-save ».

En C, on peut effectuer cette transformation *au niveau du langage source* à l'aide de **while** (true), **return** et **continue**.

En résumé

L'optimisation des appels terminaux joue un *rôle central* dans les langages fonctionnels, comme Objective Caml, où les boucles **for** et **while** ne jouent qu'un rôle secondaire.

Elle existe, avec certaines restrictions, sur la plate-forme .NET de Microsoft.

Elle *n'existe pas* en C ou Java, mais certains chercheurs se sont évertués à trouver des moyens de la simuler : voir par exemple « Tail call elimination on the Java virtual machine », de Schinz et Odersky.

De RTL à ERTL

Remarques

Optimisation des appels terminaux

4. Fonctions imbriquées et fonctions de première classe

ensiie

Appel par référence

En Pseudo-Pascal, les arguments sont passés *par valeur :* lors de l'appel f(x), la fonction f reçoit la *valeur* de x au moment de l'appel et *ne peut pas modifier* x.

Pascal et C++ permettent également le passage *par référence :* dans ce cas, lors de l'appel f(x), la fonction f reçoit *l'adresse* de x, et tout accès à x depuis f, *en lecture ou en écriture*, est traduit en un accès à la mémoire à cette adresse.

L'appel par référence est sûr, dans un langage *dénué* de fonctions de première classe, parce que la durée de vie de la variable x, qui est allouée sur la pile, *excède* la durée de l'appel à f.

Passage d'adresses en C

En C, on *simule* l'appel par référence en passant explicitement une adresse : f(x), où x est passé par référence, est simulé par f(&x), où x est passé par valeur.

Cet usage n'est *pas sûr :* rien n'empêche d'utiliser l'adresse d'une variable dont la durée de vie a expiré.

```
int* wrong ()       { int x; return &x; }
int  f     (int* x) { int y = 2; *x = 3; return y; }
int  main  ()       { printf("%d\n", f(wrong())); }
```

Fonctions imbriquées

Pascal permet la définition de fonctions locales, ou *imbriquées*, ainsi que le passage d'une fonction en tant qu'argument de fonction :

```
procedure iterate (t : array of integer; n : integer;
                   procedure f (i : integer));
var i : integer;
begin for i := 1 to n do f(t[i]) end;

function sum (t : array of integer; n : integer) : integer;
var s : integer;
   procedure add (x : integer);
   begin s := s + x end;
begin s := 0; iterate(t, n, add); sum := s end;
```

ensiie

Fonctions imbriquées

Le point-clef est que la fonction add *accède* à la variable s, laquelle est *définie par la fonction englobante* sum.

Cet accès est *sûr*, car la durée de vie de s englobe tous les points où add peut être appelée. Mais comment add obtient-elle *l'adresse* de s ?

Compilation des fonctions imbriquées

La solution usuelle est la suivante (cf. Appel, p. 133) :

- add exige en fait un *paramètre supplémentaire*, nommé lien statique (« static link ») ; il s'agit de l'adresse de la trame de pile associée à sum ;
- add *utilise* ce paramètre pour accéder à s ;
- lorsque iterate appelle add sous le nom f, il faut donc lui fournir ce paramètre supplémentaire ; iterate doit ainsi disposer non seulement de *l'adresse du code* de f mais également de son *lien statique ;*
- lorsque sum appelle iterate, il lui fournit l'adresse du code de add ainsi que l'adresse de sa propre trame de pile.

Compilation des fonctions imbriquées

Si h est imbriquée dans g qui est elle-même imbriquée dans f, alors le lien statique de h lui permet d'accéder à la trame de pile de g, donc au lien statique de g, donc à la trame de pile de f, donc aux variables locales de f.

En général, les liens statiques réalisent ainsi une *liste chaînée* de trames de pile.

Si g est récursive et imbriquée dans f, alors lorsque g appelle g, le lien statique reste inchangé, et pointe vers la trame associée à f, car les deux instances de g sont *lexicalement* imbriquées dans f.

Un lien statique peut donc pointer *arbitrairement loin* en arrière dans la pile.

Fonctions de première classe

Les langages fonctionnels, comme Haskell, SML et Objective Caml, non seulement autorisent l'imbrication de fonctions et le passage de fonctions en paramètre, mais permettent de *renvoyer* une fonction en tant que résultat, *stocker* une fonction dans une variable, etc.

Par exemple, en Objective Caml :

```
let make_cell x =
  let cell = ref x in
  let get () = !cell
  and set x = (cell := x) in
  get, set

let () =
  let get, set = make_cell 3 in set (get() + 1)
```

ensiie

Compilation des fonctions de première classe

Comme dans le cas de Pascal,

- les fonctions get et set doivent pouvoir accéder à la variable cell ; il leur faut donc un paramètre supplémentaire, nommé maintenant *environnement* ;
- une valeur fonctionnelle sera représentée par une *paire* d'un pointeur de code et d'un environnement, nommée *clôture* ;

Contrairement au cas de Pascal,

- l'environnement de get et set *n'est pas* un pointeur vers la trame de pile de make_cell, car celle-ci *disparaît* alors que get et set sont toujours susceptibles d'être appelées ;
- l'environnement de get et set est *alloué dans le tas* et contient la *valeur* de la variable cell.

ensiie

Compilation des fonctions de première classe

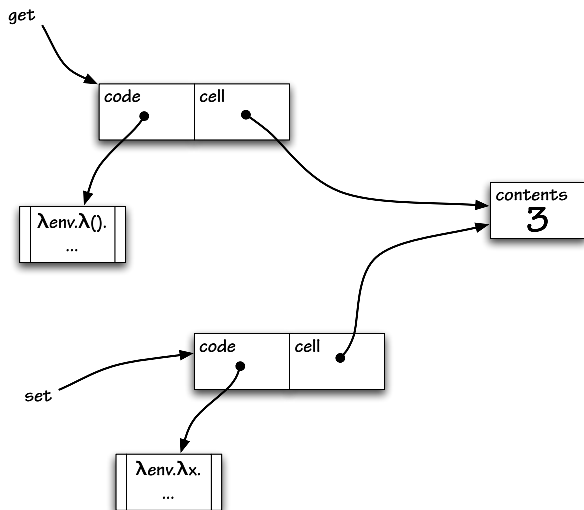
Une transformation connue sous le nom de *closure conversion* explicite ces décisions :

```
let make_cell x =
  let cell = ref x in
  let get env () = !(env.cell)
  and set env x = (env.cell := x) in
  { code = get; cell = cell }, { code = set; cell = cell }

let () =
  let get, set = make_cell 3 in set.code set (get.code get () + 1)
```

get et set sont maintenant *closes* et n'ont plus besoin d'être imbriquées dans make_cell.

Le tas après make_cell 3



ensiie

Autres schémas de compilation

Les clôtures constituent la façon habituelle de compiler les fonctions de première classe. Néanmoins, il en existe des variantes :

- la *défonctionalisation* transforme les fonctions en paires d'une *étiquette* et d'un environnement ; voir « Definitional interpreters for higher-order programming languages », de Reynolds ;
- le *λ-lifting* ajoute à chaque fonction autant de paramètres supplémentaires qu'elle a de variables libres, et s'appuie sur une notion primitive *d'application partielle* ; voir par exemple « λ-lifting in quadratic time », de Danvy et Schultz.
- certains langages, comme Scala, traduisent une fonction en un *objet* doté d'une unique méthode *apply* et dont les champs représentent l'environnement.