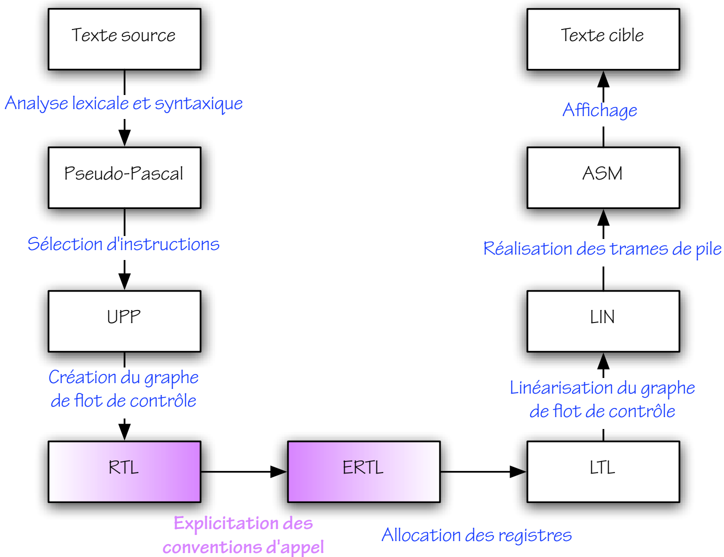
# Compilation (INF 564)

## Explicitation de la convention d'appel: de RTL à ERTL

François Pottier

20 janvier 2016


Texte source
Analyse lexicale et syntaxique
Pseudo-Pascal
Sélection d'instructions
UPP
Création du graphe
de flot de contrôle
RTL
Explicitation des
conventions d'appel
ERTL
Allocation des registres
LTL
Linéarisation du graphe
de flot de contrôle
LIN
Réalisation des trames de pile
ASM
Affichage
Texte cible

# 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.

De RTL à ERTL

Optimisation des appels terminaux

Remarques

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            (xmits $v0)
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 : ...
```

# 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
```

# 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*.

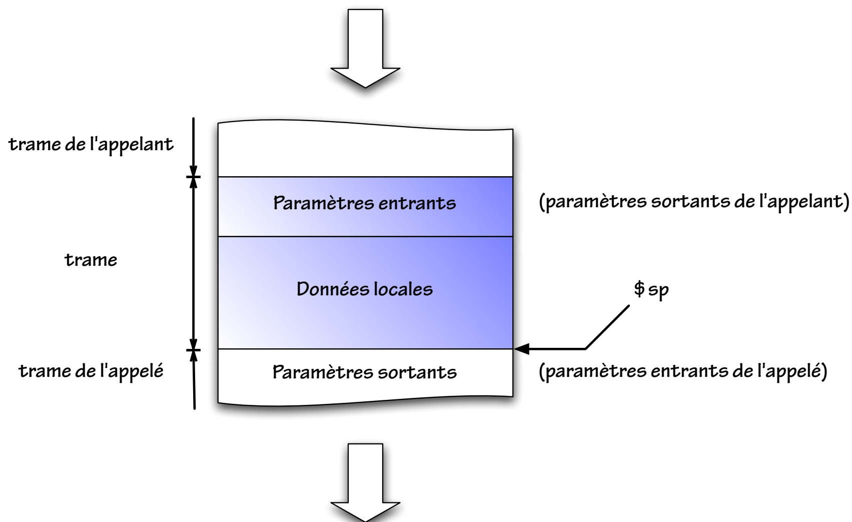
Après l'allocation de registres, chaque procédure aura besoin d'une troisième région pour contenir des données *locales*.

# 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.

# 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 ?)

# 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 est *lu a posteriori* à l'endroit convenu.

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

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

# Nouvelles instructions en ERTL (IV)

L'instruction *ICall (_, n, _)* indique combien d'arguments sont transmis via les registres \$a0, ..., $\$a_{n-1}$.

De même, *IReturn b* indique combien de résultats sont transmis. Si b vaut *true*, un résultat est transmis via \$v0; si b vaut *false*, aucun résultat n'est transmis.

Cette distinction sera exploitée pendant l'analyse de durée de vie.

# 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.

# 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           (xmits $v0)
f4 : li      %1, 1         → f0
```

# Une subtilité

Pourquoi sauvegarder/restaurer explicitement les registres callee-save ?

Ce n'est pas *nécessaire :* l'instruction **call** de ERTL *ne les écrase pas.*

Cependant, cela les rend *disponibles* ("morts") dans tout le corps de la procédure. Ils *peuvent donc servir* à réaliser des pseudo-registres !

# Une subtilité

Voici la factorielle en langage assembleur :

```
f17:
addiu $sp, $sp, -8
sw    $ra, 4($sp)
sw    $s0, 0($sp)
move  $s0, $a0
blez  $s0, f4
addiu $a0, $s0, -1
jal   f17
mul   $v0, $s0, $v0
f28:
lw    $ra, 4($sp)
lw    $s0, 0($sp)
addiu $sp, $sp, 8
jr    $ra
f4:
li    $v0, 1
j     f28
```

*On a sauvegardé/restauré* les callee-save en passant de RTL à ERTL.

# Une subtilité

*Et si on ne les sauvegarde/restaure pas* en passant de RTL à ERTL :

```
f9:                          lw     $v1, 0($sp)
addiu $sp, $sp, −8           mul    $v0, $v1, $v0
sw    $ra, 4($sp)            f12:
sw    $a0, 0($sp)            lw     $ra, 4($sp)
lw    $v1, 0($sp)            addiu  $sp, $sp, 8
blez  $v1, f4                jr     $ra
lw    $v1, 0($sp)            f4:
addiu $a0, $v1, −1           li     $v0, 1
jal   f9                     j      f12
```

Même espace réservé sur la pile, mais *4* ***lw*** *au lieu de 2.*

De RTL à ERTL

## Optimisation des appels terminaux

Remarques

Fonctions de première classe

# 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.

# 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 résumé

L'optimisation des appels terminaux joue un *rôle central* dans les langages fonctionnels, comme Scheme, Haskell, ou OCaml, 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

Optimisation des appels terminaux

## Remarques

Fonctions de première classe

# 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 ?

# 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.

# 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.

De RTL à ERTL

Optimisation des appels terminaux

Remarques

Fonctions de première classe

# Fonctions imbriquées

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

```
let iter f t =
  for i = 0 to Array.length t - 1 do f t.(i) done

let sum t =
  let s = ref 0 in
  let add x =
    s := !s + x
  in
  iter add t;
  !s
```

Pascal le permet également.

# Fonctions imbriquées

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

*Comment* add obtient-elle l'adresse de s ? En effet, s n'est ni un de ses paramètres, ni une de ses variables locales, ni une variable globale.

# Fonctions de première classe

En OCaml (*et en* SML, Haskell, Scala, ...), *on* peut non seulement imbriquer *des* fonctions *et* passer une fonction *en* paramètre, mais aussi :

- *renvoyer* une fonction en tant que résultat,
- *stocker* une fonction dans le tas.

La *durée de vie* d'une fonction peut alors *dépasser* la durée de vie des variables locales des fonctions englobantes !...

# Fonctions de première classe

Exemple : une cellule accessible uniquement via deux fonctions get/set.

```
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)
```

*Comment* get et set obtiennent-elles l'adresse de la cellule ?...

La variable locale cell de la fonction make_cell *n'existe plus* quand get et set sont appelées !

# Compilation des fonctions de première classe

Le principe général est simple et ancien (Scheme, 1975–78) :

- get et set doivent accéder à la *valeur* de la variable cell ; il leur faut donc un paramètre supplémentaire, *l'environnement*, qui doit permettre d'accéder à cette valeur ;
- une fonction est représentée par une *clôture*, qui doit permettre d'accéder au *code* de la fonction et à son *environnement* ;
- clôture et environnement doivent être *alloués dans le tas*, car leur durée de vie est non bornée.

(Ici, environnement et clôture forment un seul bloc de mémoire.)

# Compilation des fonctions de première classe

Une transformation nommée *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* (sans mention de variables externes) et n'ont donc plus besoin d'être imbriquées dans make_cell.

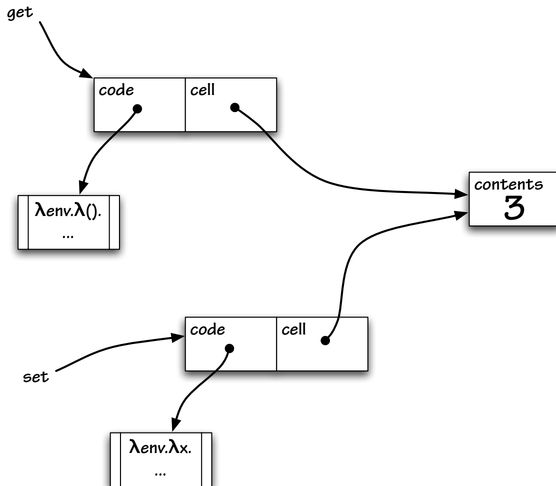
# Clôtures

{ code = get; cell = cell } est une *clôture*.

C'est un enregistrement (ou *record*) dont :

- le champ code contient une fonction close;
- les autres champs, ici cell, contiennent les valeurs dont cette fonction a besoin.

# Le tas après make_cell 3



# En résumé

La closure conversion permet de traduire :

- un langage avec fonctions de première classe *et fonctions imbriquées*, comme OCaml, vers :
- un langage avec fonctions de première classe mais *sans fonctions imbriquées*, comme C.

# Autres schémas de compilation

Il existe d'autres moyens d'effectuer une telle traduction :

- 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.