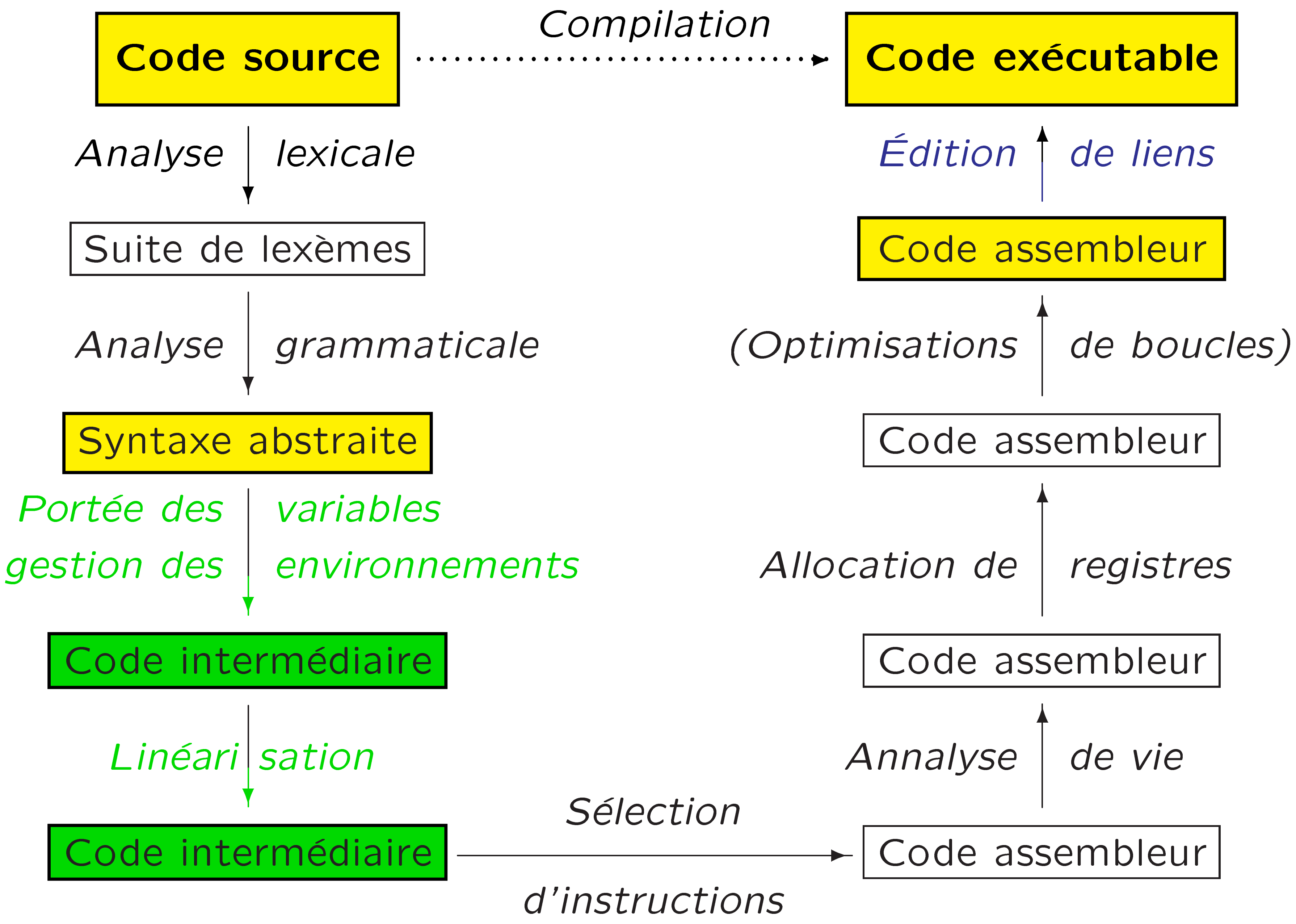
Code Intermédiaire
Génération de Code
Linéarisation
Canonisation

Didier Rémy

Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/3/
http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/3/


Code source
Compilation
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection
d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens

Difficulté de la génération de code

- Gestion des environnements : conventions d'appel des fonctions, sauvegarde en pile, etc.
- Gestion des registres.
- Mécanisation des optimisations.

Code Intermédiaire

Découpage en petites étapes

$$\text{SA} \underbrace{\overset{(1)}{\xrightarrow{\text{Générer du code}} \text{CI} \xrightarrow{\text{Canoniser Linéariser}}}}_{\text{Code intermédiaire linéaire, inefficace}} \text{CI} \underbrace{\overset{(2)}{\xrightarrow{\text{Sélection d'instruct.}} \text{CA} \xrightarrow{\text{Allocation de registres}}}}_{\text{Code machine Efficace}} \text{CA}$$

Approche systématique : robuste, modulable, réutilisable.

Approche globale :

- La phase (1) introduit des sources d'inefficacités
- éliminées par la phase (2).

Pourquoi un code intermédiaire ?

Partage du travail

– Entre plusieurs langages, et plusieurs machines

$$\left.\begin{array}{r}\text{Java}\\ \text{C}\\ \dots\\ \text{Caml}\\ \text{Ocaml}\end{array}\right\} \quad \boxed{\text{CI}} \quad \left\{\begin{array}{l}\text{MIPS}\\ \text{xx86}\\ \dots\\ \text{Sparc}\\ \text{Byte-code}\end{array}\right.$$

– Entre les différentes constructions d'une même langage ou d'une même machine
  – La syntaxe abstraite comporte trop de constructions voisines
  – Le code assembleur comporte trop d'instructions voisines

La représentation intermédiaire est plus concise.

Le principe du code intermédiaire

Le but du code intermédiaire est de passer d'une structure arborescente à une (bonne) structure linéaire et de préparer la sélection d'instructions.

Les détails dépendant de l'architecture sont relégués à une phase ultérieure de sélection d'instructions.

Quelques caractéristiques du code intermédiaire :

– Les branchements sont explicites.
– Code arborescent (expressions) ou linéaire (instructions)
– Utilise une infinité de registres (temporaires), dont l'utilisation est privilégiée (réversible) et le coût négligé.
– L'adressage en mémoire est une forme séparée qui n'est retenue que lorsque c'est indispensable (irréversible).
– L'appel de fonction est implicite, et sera résolu dans une phase ultérieure.
  (Différentes formes dans différents langages d'assemblage)

Les expressions (Code.exp)

```
type exp =
     Const of int                     (* Entiers  et  Booléens *)
   | Name of label                    (* Globaux *)
   | Temp of temp                     (* Lecture d'un temporaire *)
   | Mem of exp                       (* Lecture mémoire *)
   | Bin of binop * exp * exp         (* Opération binaire  *)
   | Call  of frame * exp list        (* Appel de fonction *)
```

Pour certains langages, il est utile d'ajouter un nœud

```
   | Eseq of stm * exp
```

pour représenter une séquence retournant un résultat (comme `e1; e2` *en ML). Ce n'est pas nécessaire pour Pseudo-Pascal.*

Les instructions (Code.stm)

```
and stm =
  | Label of label                    (* Étiquette *)
  | Move_temp of temp * exp           (* Écriture de exp dans un
                                         temporaire *)
  | Move_mem of exp * exp             (* Écriture en mémoire à
                                         une adresse calculée *)
  | Seq of stm list                   (* Séquence d'instructions *)
  | Exp of exp                        (* Expression avec effet *)
  | Jump of label
  | Cjump of relop * exp * exp * label * label

and relop = Req | Rne | Rle | Rge | Rlt | Rgt
and binop = Plus | Minus | ... | Eq | Ne
```

Utilisation des temporaires

Les étiquettes et les temporaires sont créés à la demande. Par sécurité on les manipulera de façon abstraite. Penser

temporaire = registre virtuel

Les temporaires seront associés à des registres réels, et éventuellement à des emplacements en pile pour les sauvegarder.

La génération de code crée beaucoup de temporaires, mais de durée de vie très courte. Les temporaires avec une durée de vie très courte n'auront pas besoin d'être sauvegardés en pile.

Plus on en crée, plus leur durée de vie sera courte et plus facilement ils pourront partager le même registre : ne jamais chercher à réutiliser le même temporaire, au contraire utiliser des temporaires différents dès que possible !

Première partie
Génération de code

Génération de code

On traduit récursivement expressions et instructions :

$[\![\_]\!]^e$: Pp.expression $\rightarrow$ Code.exp

$[\![\_]\!]^s$: Pp.instruction $\rightarrow$ Code.stm

de façon la plus simple possible.

$[\![\texttt{Int}\ n]\!]^e = \texttt{Const}\ n \quad [\![\texttt{Bool}\ true]\!]^e = \texttt{Const}\ 1 \quad [\![\texttt{Bool}\ false]\!]^e = \texttt{Const}\ 0$

$$
\begin{array}{rcl}
[\![\texttt{Bin}\,(op, e_1, e_2)]\!]^e & = & \texttt{Bin}\,(op, [\![e_1]\!]^e, [\![e_2]\!]^e) \\
[\![\texttt{Sequence}\,(s_1; \ldots; s_n)]\!]^s & = & \texttt{Seq}\,([\![s_1]\!]^s; \ldots; [\![s_n]\!]^s)
\end{array}
$$

Opérations sur les tableaux (alloués dans le tas) :

$$
\begin{array}{rcl}
[\![\texttt{Geti}\,(e_1, e_2)]\!]^e & = & \texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e, [\![e_2]\!]^e)) \\
[\![\texttt{Seti}\,(e_1, e_2, e_3)]\!]^s & = & \texttt{Move\_mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, [\![e_1]\!]^e, [\![e_2]\!]^e), [\![e_3]\!]^e)
\end{array}
$$

Expressions conditionnelles

Test simple :

$$[\![\texttt{If}\,(\texttt{Bin}\,(op, e_1, e_2), s_t, s_f)]\!]^s = \quad \texttt{Seq}\left[\begin{array}{l} \texttt{Cjump}\,(op, [\![e_1]\!]^e, [\![e_2]\!]^e, l_t, l_f); \\ \texttt{Label}\ l_t; [\![s_t]\!]^s; \texttt{Jump}\ fi; \\ \texttt{Label}\ l_f; [\![s_f]\!]^s; \texttt{Jump}\ fi; \\ \texttt{Label}\ fi; \end{array}\right]$$

Test complexe :

$$[\![\texttt{If}\,(e_1, s_t, s_f)]\!]^s = [\![\texttt{If}\,(\texttt{Bin}\,(\texttt{Ne}\,, e_1, \texttt{Const}\ 0), s_t, s_f)]\!]^s$$

La boucle while se traite de façon similaire.

Gestion des environnements

Pour comprendre la traduction des variables, il faut comprendre comment sont traités les arguments et les variables locales des fonctions.

Cela dépend des constructions du langage.

Par exemple :

- L'appel par référence (ou `&x` en C) permet de passer l'adresse plutôt que la valeur d'une variable. Ainsi, une variable contient tantôt la valeur cherchée tantôt l'adresse à laquelle chercher cette valeur.
- Les tableaux en pile sont (en général) passés par référence.
- Les fonctions locales obligent à passer par référence soit :
  - les variables qui s'échappent dans la fonction locale (remontée des variables),
  - le lien statique (fermeture en pile $\approx$ tableau en pile).

Cas simplifié de Pseudo-Pascal

En Pseudo-Pascal, il n'y a pas de fonctions locales,

– les arguments sont passés par valeurs, (i.e. le contenu des variables et non la boîte).
– les tableaux sont alloués dans le tas et passés par référence (e.g. passe leur adresse mais on ne les recopie pas).

En particulier, une variable ne peut pas être directement affectée par un appel de fonction (mais la partie mutable d'une valeur associée à une variable peut être modifiée).

On place les variables dans des temporaires (qui en général seront des registres).

On accède à (on écrit dans) à une variable en lisant (en écrivant dans) le temporaire associé.

Compilation des accès

L'accès à une variable locale est donc toujours la lecture d'un temporaire. Son numéro est trouvé dans l'environnement de compilation (cf. analogue à l'environnement d'évaluation) qu'il faut passer comme argument à la fonction de compilation.

$$[\![\texttt{Get}\ x]\!]^e{}_\rho = \texttt{Temp}\ t \qquad \text{si } \rho(x) = t$$

(On remplace partout ailleurs $[\![e]\!]^e$ et $[\![e]\!]^s$ par $[\![e]\!]^e{}_\rho$ et $[\![e]\!]^s{}_\rho$.)

Les globaux sont mémorisés dans un tableau statique alloué au début du programme à l'adresse symbolique G. Dans ρ les globaux sont définis par leur position dans le tableau (w est la taille du mot).

$$[\![\texttt{Get}\ x]\!]^e{}_\rho = \texttt{Mem}\,(\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Name}\ G,\texttt{Const}\,(n * w))) \qquad \text{si } \rho(x) = n$$

On pourrait aussi placer `Name` G dans un temporaire spécial réservé `Temp` t_G.

Mise en œuvre

Dans l'environnement de compilation, il faut associer aux variables (chaînes de caractères) des emplacements.

Ce dont sont des temporaires pour les variables locales, soit un indice dans la table globale. On utilise un type somme :

```
type access = Local of Gen.temp | Global of int
let rec translate_expr env = function
    ...
  | Get x ->
     begin match Env.find_var env x with
     | Local t -> Temp t
     | Global n ->
        let g = Frame.global_space and w = Frame.word_size
        in Mem (Bin (Name g , Const (n * w)))
     end
```

Compilation des affectations

De façon analogue, c'est l'écriture :

- dans le temporaire associé à la variable considérée, pour les accès locaux ;
- dans la case du tableau des globaux associé à la variable considérée.

Formellement :

$$[\![\texttt{Set}\ x\ e]\!]^s{}_\rho =$$

$$\begin{array}{ll} \texttt{Move\_temp}\,(t, [\![e]\!]^e{}_\rho) & \text{si } \rho(x) = t \\ \texttt{Move\_mem}\,(\texttt{Bin}\,(\texttt{Plus}\,, \texttt{Name}\ G, \texttt{Const}\,(n * w)), [\![e]\!]^e{}_\rho) & \text{si } \rho(x) = n \end{array}$$

Utilisation la pile en Pseudo-Pascal

La pile ne sert plus qu'à passer les arguments (> 4) et à sauvegarder les registres pendant les appels récursifs, ou lorsqu'il y a trop de temporaires.

Ces utilisations de la pile seront traités ultérieurement par la compilation des appels de fonction et par l'allocateur de registres.

En effet, dans le code intermédiaire,
- On utilise un nombre arbitraire de temporaires, que l'on suppose sauvegardés pas les appels de fonctions.
- On utilise une instruction de haut niveaux pour les appels de fonctions ; elle devra être remplacée ultérieurement par du code qui place les arguments à des positions conventionnelles et sauvergarde, si nécessaire, les valeurs des temporaires pendant les appels (e.g. récursifs).

Compilation des fonctions

Lors de l'appel de fonction, il faut communiquer entre l'appelant et l'appelé pour passer les arguments et recevoir les résultats. Par exemple les premiers arguments sont passés dans les registres a_0 à a_3, les suivants sur la pile.

– Avant l'appel les arguments sont dans des temporaires de l'appelant.
– Après, ils sont dans des temporaires de l'appelé, a priori différents (sauf avec allocation de registres inter-procédurale).

L'interface entre la vue externe et la vue interne, i.e. le (code de) transfert des arguments entre l'appelant et l'appelé, sera inséré ultérieurement en fonction de la machine cible, lors de la traduction des instructions `Call` et en ajoutant un prologue et un épilogue à chaque fonction.

L'unité de traduction en code intermédiaire est la fonction ou procédure.

Exemple

Fonction succ en Pseudo-Pascal (syntaxe abstraite) :

```
{ arguments = [ "n", Integer ];    result  = Some Integer;
  local_vars  = [  ];
  body = Set ("succ", Bin (Plus, Get "n", Int 1)) }
```

Son code intermédiaire :

```
[ Label succ;
  Move_temp
    (t104, Code.Bin (Code.Plus, Temp t105, Const 1));
  Jump succ_end ]
```

où

t104	emplacement de l'argument `n`
t105	emplacement du résultat de `succ`
succ	adresse du prélude (utilisée pour l'appel)
succ_end	adresse du prologue (utilisée pour le retour)

Exemple (suite)

Appel de la fonction `succ` ;

```
Function_call ("succ", Const 2)
```

Code de l'appel :

```
Call (<succ>, [Const 3])]
```

C'est à la charge de l'instruction Call de placer les arguments dans le temporaire `t104` et de récupérer le résultat dans le temporaire `t105`.

Cela sera réalisé dans une phase ultérieure de la compilation, après l'allocation des registres, par le prélude et le prologue.

Compilation des appels

L'appel de fonction est compilé par une expression Call

$$[\![\texttt{Function\_call}\,(g, e_1, \ldots, e_n)]\!]^e = \texttt{Call}\,(F_g, [\![e_1]\!]^e, \ldots, [\![e_n]\!]^e)$$

où F_g est une structure décrivant g (appelée frame) comprenant :

- L'étiquette pour entrer dans la fonction (celle placée en tout début du code de g). Plus tard, on y insérera un prélude (avec la sauvegarde éventuelle de certains registres).
- Une étiquette pour sortir de la fonction. Plus tard on y insérera le prologue (restauration des registres)
  Pour sortir d'une fonction il faut toujours faire un saut au prologue. (c'est convention que nous prenons à ce stade.)
- L'emplacement exact des arguments dans la vue interne : position en pile, ou numéro de temporaire (il s'agit toujours d'un temporaire en Pseudo-Pascal).
- Position du résultat (si c'est une fonction).

Mise en œuvre

Le type frame est implémenté dans frame.ml

```
type frame =
    { name : Gen.label;
      return_label : Gen.label;
      args : address list ;
      result : address option;
      mutable locals : int ; }
```

Mais, on les manipule de façon abstraite (interface frame.mli) pour permettre une modification ultérieure si nécessaire.

Le frame est recherché dans l'environnement de compilation :

```
| Function_call (f, args) ->
    let frame_f = Env.find_definition env f in
    let args_compilés = List.map (translate_expr env) args
    in Call (g, args_compilés)
```

Compilation du corps des fonctions

Pour chaque fonction :

1. Créer un nouveau frame, avec des emplacements internes pour les arguments, et le résultat (optionnel).
   (Ce sont simplement des temporaires –fraîchement alloués– en pseudo-pascal)
2. Attribuer des emplacements aux variables locales (idem)
3. Augmenter l'environnement de compilation.
4. Compiler le corps dans cet environnement.

Récursion

Comme les fonctions sont récursives en PP, l'étape 1 (allocation des frames) est faite en premier pour toutes les définitions, avant d'effectuer les autres phases pour chaque définition.

Compilation dans le cas général

Pseudo-pascal est un cas simplifié :
- Les arguments sont passés par valeurs.
- Les tableaux sont passés par références.

Variables passées par référence : Il faut distinguer l'adressage direct et indirect (i.e. Left- et Right-value).

Tableaux alloués dans la pile : Idem. De plus, toutes les variables allouées n'ont plus la même taille (ce qu'il faut gérer).

Fonctions locales (non retournées comme valeur) Au choix :
- chaîner les blocs d'activation en pile (cf liens statiques), ou
- remonter les variables pour rendre toutes les fonctions globales.

Les variables qui s'échappent sont toujours placées en pile.

Fonctions comme valeur (ML) : fabriquer des fermetures.

Liens statiques

Pour les fonctions locales non retournées, la fermeture peut être allouée en pile. Pour simplifier, supposons que toutes les variables de f s'échappent, par exemple, elles sont utilisées par une fonction g locale à f, et sont placées en pile (x_1, ... x_n) :

	$\vdots$	
pointeur de bloc $\rightarrow$	ℓ	Lien statique
	x_1	
	$\vdots$	Variables de f
	x_n	
pointeur de pile $\rightarrow$	$\vdots$	Registres sauvegardés
$\downarrow\uparrow$	$\vdots$	autres appels
		appel à g

Explication par remontée des variables

Par remontée des variables il faut passer à g en plus de ses arguments $y_1, \ldots y_q$, les adresses $\&x_1, \ldots \&x_q$ des variables de f.

À condition de figer statiquement leur position en pile, il suffit de passer à g l'adresse du bloc d'activation de f (l'environnement dynamique dans lequel est défini g) qui marque le début des variables de f. Vu de g, on l'appelle le **lien statique de** g.

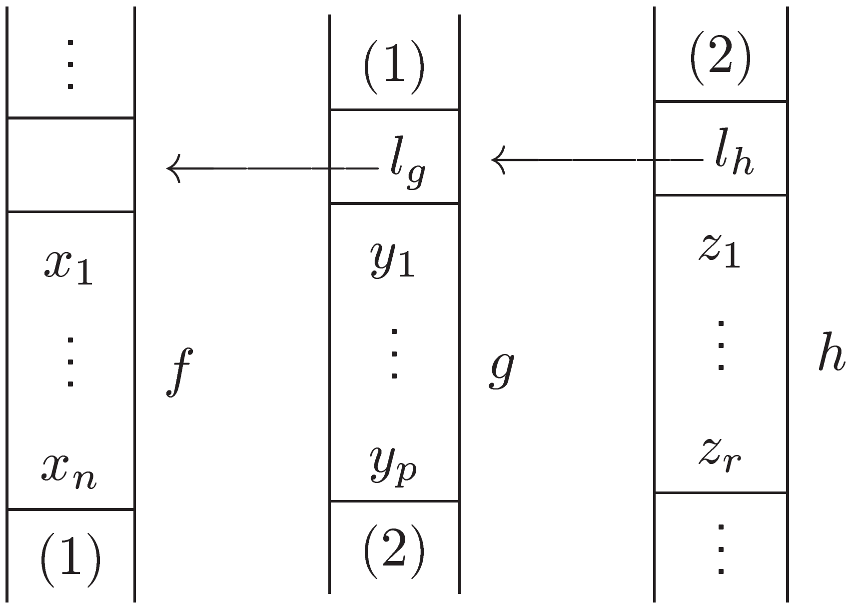
La fonction g retrouve les x_i par décalage (calculer statiquement) par rapport au lien statique (passé dynamiquement).

- elle est compilée comme si elle recevait en plus de ses arguments un enregistrement ℓ_g dont les champs sont x_1, .. x_n.
- elle est appelée en passant en plus des arguments y_1, ... y_q le lien statique ℓ_g.

Vu comme une fermeture en pile

Au lieu d'être représentés à plat les environnements sont vus comme une chaîne de petits environnements à plat.

Il peuvent être directement implémentés dans la pile tant que les fonctions ne sont pas retournées en résultat (leur durée de vie dynamique n'excède par leur portée lexicale).



L'unité de compilation

Par simplicité, l'unité de compilation est la procédure : nous ne ferons pas ici d'optimisations ni d'allocation de registres inter-procédurale.

Chaque unité de compilation comporte un frame (bloc d'activation).

On traite le corps principal de la même façon, en le transformant en une procédure qui ne reçoit pas d'arguments et sans variables locales.

Les optimisations inter-procédurales sont possibles, mais plus complexes et plus coûteuses.

Le programme principal

Les étapes

– Décider des emplacements pour les globaux,
– Allouer les frames des fonctions globales,
– Créer l'environnement global,
– Compiler le corps des procédures (y compris le corps du programme)

Le résultat de la compilation : une structure décrivant :

– le nombre de globaux.
– la procédure principale (distinguée) et les sous-procédures, chacune comportant un frame et son code (intermédiaire).

```
type 'a procedure = Frame.frame * 'a
type 'a program =
    { number_of_globals : int;
      main : 'a procedure; procedures : 'a procedure list }
val program : Pp.program -> Code.code program
```

Les primitives

Dans le code intermédiaire, on traite les primitives comme des appels à des fonctions externes.

Il suffit donc de se donner une étiquette réservée par primitive (en fait un frame dont seule l'étiquette importe).

$$[\![\texttt{Alloc}\ e)]\!]^e = \texttt{Call}\,(alloc, [\![e]\!]^e)$$

où $alloc$ est le frame de la primitive `alloc` (créé une seule fois).

Le fichier frame.ml contient les déclarations :

```
let write_int   = Frame.new_primitive "write_int"   1 false
let writeln_int = Frame.new_primitive "writeln_int" 1 false
let read_int    = Frame.new_primitive "read_int"    0 true
let alloc       = Frame.new_primitive "alloc"       1 true
```

(le premier argument indique l'arité de la primitive, le second si elle retourne ou non un résultat.)

Linéarisation

Linéarisation (but)

- Les expressions permettent des structures emboîtées telles que $\texttt{Call}\,(f, e_1) + \texttt{Call}\,(g, (\texttt{Call}\,(h, e_2))))$. Or, un appel de fonction est une opération complexe, qui produit des effets de bords (écriture dans des registres).
- Dans la plupart des langages (mais pas en Pseudo-Pascal) les expressions peuvent contenir des séquences. Par exemple $e_1 + (s; e_2)$ en ML. À nouveau s peut modifier la valeur des registres utilisés pour évaluer e_1.

La linéarisation est une phase postérieure à la génération de code qui :

- extrait les appels de fonctions et les séquences des expressions.
- aplatit les séquences d'instructions.

Exemple

Le code suivant est non linéaire (il contient des instructions Call dans les expressions).

$$\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Call}\,(f,e_1),\texttt{Call}\,(g,(\texttt{Call}\,(h,e_2))))$$

La linéarisation le transforme en :

$$\begin{array}{l}
\texttt{Move}\,(\texttt{Temp}\ t_1,e_1);\\
\texttt{Move}\,(\texttt{Temp}\ t_2,\texttt{Call}\,(f,t_1));\\
\texttt{Move}\,(\texttt{Temp}\ t_3,e_2);\\
\texttt{Move}\,(\texttt{Temp}\ t_4,\texttt{Call}\,(h,t_3));\\
\texttt{Move}\,(\texttt{Temp}\ t_5,\texttt{Call}\,(g,t_4)),\\
\texttt{Bin}\,(\texttt{Plus}\,,\texttt{Temp}\ t_1,\texttt{Temp}\ t_5)
\end{array}$$

Linéarisation (méthode)

Par réécriture du code. La règle principale d'extraction

$$\texttt{Call}\,(f, e_1, e_2) \longrightarrow \texttt{Move\_temp}\,(t, \texttt{Call}\,(f, e_1, e_2)), \texttt{Temp}\ t$$

Plus généralement l'extraction retire d'une expression e une séquence s en fournissant une expression résiduelle e'. Elle procède récursivement pour atteindre les expressions les plus internes :

$$\frac{e_1 \longrightarrow s, e_1'}{e_1 + e_2 \longrightarrow s, (e_1' + e_2)} \qquad \frac{e_2 \longrightarrow s, e_2' \qquad \boxed{(e_1, s) \text{ commutent}}}{e_1 + e_2 \longrightarrow s, (e_1 + e_2')}$$

Pour préserver l'ordre d'évaluation, il faut aussi extraire les expressions qui se trouve sur le chemin et qui ne commutent pas :

$$\frac{e_2 \longrightarrow s, e_2'}{e_1 + e_2 \longrightarrow \texttt{Move\_temp}\,(t, e_1); s, (\texttt{Temp}\ t + e_2')}$$

(Règle à n'appliquer qu'en dernier recours.)

Commutation

Il s'agit de ne pas permuter l'évaluation d'une instruction et d'une expression lorsque cela risque de changer le sens du programme.

– typiquement, par le changement de l'ordre lecture-écriture,
– éventuellement, par la levée d'une exception.

Une assez bonne approximation est qu'une expression constante qui ne lit ni n'écrit commute avec toute instruction.

(A priori une instruction extraite écrit quelque chose, donc le cas sans écriture qui commute évidemment n'est pas intéressant.)

Pour être plus fin, on peut autoriser des lectures et écritures dans des emplacements (temporaire ou mémoire) disjoints.

Linéarisation (suite)

On procède de même pour toutes les expressions pouvant contenir des expressions complexes (call) aux feuilles, et pour les instructions qui peuvent contenir des expressions...

$$\frac{e_1 \longrightarrow s, e_1'}{\mathtt{Exp}\,(e_1) \longrightarrow s; \mathtt{Exp}\ e_1'} \qquad \frac{i \longrightarrow s'; i'}{s; i \longrightarrow s; s'; i'} \qquad \textit{etc.}$$

On en profite pour aplatir les séquences

$$\frac{s \longrightarrow s'}{s_0; \mathsf{Seq}\,(s) \longrightarrow s; s'}$$

Attention à ne pas extraire les expressions déjà extraites, i.e. les `Call` qui apparaissent déjà dans l'une des deux configurations :

$$\mathtt{Move}\,(\mathtt{Temp}\ t, \mathtt{Call}\,(f, \mathtt{Temp}\ t_i)) \qquad \text{ou} \qquad \mathtt{Exp}\,(\mathtt{Call}\,(f, \mathtt{Temp}\ t_i))$$

Mise en œuvre de la linéarisation

Les fonctions principales rewrite_stm et rewrite_exp décrivent les règles de réécriture en utilisant le pattern matching.

```
let rec rewrite_exp : exp -> stm list * exp =
  function
    | Call (f, el) ->
        let s, el' = rewrite_args el in
        let t = G.new_temp() in
        s @ [Move_temp (t, Call (f, el' ))], Temp t
    | Bin (binop, e1, e2) ->
        let s, [ e1';e2'] = rewrite_args args in
        s, Bin (binop, e1', e2')
    | .......
```

Mise en œuvre (suite)

```
and rewrite_stm : stm -> stm list =
  function
    | Move_temp (t, Call (f, el)) ->
        let s, el'  = rewrite_args el in
        s @ [Move_temp (t, Call (f, el' ))]
    | Move_temp (t, e1) ->
        let s, e1' = rewrite_exp e1 in
        s @ [Move_temp (t, e1')]
              ......
```

La procédure auxiliaire `rewrite_args` réécrit une liste d'expressions (arguments) :

```
and rewrite_args : exp list -> stn list * exp list = ...
```

(Elle doit gérer les problèmes de commutation)

Blocs élémentaires
Calcul des traces

Les blocs de base

Un bloc de base est une séquence d'instructions linéaire (une seule entrée, une seule sortie) la plus longue possible. Il commence par une étiquette, fini par un saut et ne comporte ni étiquette (point entrée) ni saut (point de sortie) au milieu.

Le calcul des blocs de base est immédiat : il suffit de parcourir le code en commençant par le début, et chaque instruction de saut ou étiquette termine le bloc courant et en commence un nouveau.

On ajoute si nécessaire des instructions Jump lorsque le bloc courant est fermé par la rencontre d'une étiquette.

Les traces

Une trace d'exécution est un parcours le plus long possible en enchaînant les blocs de bases les uns à la suite des autres sans jamais repasser par un bloc déjà vu.

Cela revient à un parcours de graphe orienté où les sommets sont les blocs de base et les arcs relient deux blocs lorsqu'il existe un saut du premier vers le second.

On visite alors tous les noeuds du graphe, en allant le plus loin possible sans jamais repasser par un nœud déjà vu.

Nettoyage des traces

Les traces mettent bout à bout des sauts et le bloc suivant. On trouve donc fréquemment la configuration suivante que l'on simplifie (économiser des sauts étant un des buts recherchés) :

$$\texttt{Jump } l; \texttt{Name } l \longrightarrow \texttt{Name } l$$

Les instructions de saut conditionnel de tous les assembleurs prennent l'instruction qui suit immédiatement pour branche "false". On réarrange les sauts conditionnels tels qu'ils soient toujours suivis par leur branche false, soit en inversant les tests (suffit en général) soit en introduisant une étiquette intermédiaire.

$$\texttt{Cjump}\,(op, e_1, e_2, l_1, l_2); \texttt{Name } l_1 \longrightarrow \texttt{Cjump}\,(neg(op), e_1, e_2, l_2, l_1); \texttt{Name } l_1$$

$$\texttt{Cjump}\,(op, e_1, e_2, l_1, l_2) \longrightarrow \texttt{Cjump}\,(op, e_1, e_2, l_1, l_3); \texttt{Name } l_3; \texttt{Jump } l_2$$

Traces optimales

Certaines traces sont meilleures que d'autres.

En effet, il est plus important de minimiser les sauts dans le code fréquemment parcouru, par exemple dans les boucles les plus internes.

Une telle boucle devra de préférence être toute entière dans sa trace. (Un seul saut pour le retour dans le cas normal).

Une meilleure approximation consisterait à rechercher les boucles les plus internes par un tri topologique sur les blocs de base. Puis à commencer par linéariser dans l'ordre de domination.

Calcul incrémental des traces

Dans un langage structuré (sans instruction goto), il n'est pas possible d'entrer dans un bloc “par le milieu”. On peut se passer du du calcul des traces, et généré dès le départ du code linéaire canonique...

Le résultat est comparable, voir meilleur qu'un calcul des traces automatique mais naïf, mais moins bon qu'un calcul qui utilise les dominateurs.

De plus, l'approche est moins modulaire, car le calcul des traces est est délocalisé dans la compilation de chaque construction de branchement, alors que dans l'approche générale il ne dépend que du code intermédiaire et est indépendant de la compilation de la syntaxe abstraite en code intermédiaire.

Exercices

Exercice 1 (Génération du code intermédiaire) *Écire un générateur de code intermédiare pour Pseudo-Pascal. Le code intermédiaire est donné par l'interface code.mli.*

La gestion des étiquettes, des environnements et des frames sont données par les fichiers env.mli, gen.mli et frame.mli. □

Exercice 2 (Linéarisation et canonisation) *Implémenter la linéarisation, le calcul des blocs de base et la mise sous forme canonique.*

On impose l'interface canon.mli. □

Exercice 3 (Interprète du code intermédiaire) *Écire un interprète pour le code intermédiaire. Celui-ci devra fonctionner aussi bien sur du code arborescent que sur du code linéaire. On pourra également écrire un imprimeur du code intermédiaire.* □

Génération de code intermédiaire

Librairies fournies

- La syntaxe abstraite de Pseudo-Pascal pp.mli
- La gestion des environnements env.mli et env.ml
  (Reprend les fonctions écrites au td2 pour l'interprète.)
- La gestions des étiquettes et temporaires gen.mli et gen.ml.

```
type temp
val new_temp : unit -> temp

type label
val new_label: unit -> label
val named_label : string -> label
```

– La gestion des blocs d'activation frame.ml et frame.mli

```
type frame
val new_frame : var_list  -> type_expr option -> frame
val named_frame
    : string  -> var_list  -> type_expr option -> frame
val frame_name : frame -> label
val frame_result  :  frame -> temp option
val frame_return :  frame -> label
```

– La définition du code intermédiaire code.mli

Pour récupérer le tout dans le répertoire courant (attention à vous placer au bon endroit) :

```
cp ~remy/compil/td3/* ./
```

On vous donne une implémentation très incomplète trans.ml et son interface trans.mli.

Installation de l'ensemble

Vérifier d'abord que l'ensemble est bien installé. La version incomplète de `trans.ml` est compilable.

On pourra utiliser ce Makefile et exécuter

```
make
ocaml un.mlx
```

Pour voir le code produit

```
ocaml < un.mlx
```

(et voici le fichier `run.ml`)

Compléter le fichier trans.ml

On traitera d'abord les expressions et les instructions simples, puis les variables globales, afin de traiter le programme pascal `deux.p`, i.e. :

```
ocaml deux.mlx
```

Compilation des fonctions

Écrire la compilation des fonctions.

(On pourra reprendre la compilation du corps d'un programme pour le traiter comme celui d'une procédure sans arguments et sans variables locales).

On pourra tester avec les programmes `fact.mlx` et `fib.mlx` (voir le code source et les entrée à fournir dans le cours précédent

Compilation des tableaux

On pourra enfin tester avec les programmes "tri.mlx" "quick.mlx" et "bignum.mlx".

Solution : trans.ml

Note : Les fichiers d'interfaces `*.mli` utilisés par `trans.ml` comportent maintenant une spécification plus détaillée des opérations implémentées.

Interprète du code intermédiaire

Le simulateur fourni est un interprète le code intermédiaire, qui fonctionne avant ou après la linéarisation du code.

Cet interprète est donné comme un module `simul.ml` d'interface `simul.mli`, compatible avec les conventions de `frame.mli` :

- L'appel d'une fonction ou d'une procédure, ainsi que du programme principal s'effectue en sautant à l'étiquette du frame correspondant.
- Le retour des même appels est détecté par un saut au prologue du frame.
- Les sauts aux adresses des primitives déclarées dans `frame.mli` sont reconnues les primitives correspondantes sont exécutées.
- Les globaux commencent à l'adresse `global_space`.

Une différence importante par rapport à une machine réelle, est que l'interprète sauvegarde tous les temporaires avant un appel les restaure au retour.

Canonisation du code

Pour ceux qui le souhaite, vous pouvez implémenter la canonisation et la linéarisation du code obtenu par la première étape de la compilation.

Interpafe imposée canon.mli

Solution canon.ml

(Utilise de façon optionnel un module scc.ml d'interface scc.mli qui énumère les composantes connexes dans un ordre de non dépendance).