

Chapitre 6

Traduction en code intermédiaire

Sommaire

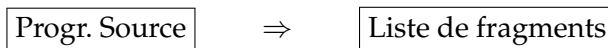
6.1	Le code intermédiaire	1
6.1.1	Preliminaires	1
6.1.2	Description du code intermédiaire	2
6.1.3	Traduction en code intermédiaire	4
6.1.4	EXP et ESEQ	6
6.1.5	Structure de la fonction de traduction	8
6.2	Transformations du code intermédiaire	9
6.2.1	Une pause de réflexion	9
6.2.2	Mise en forme canonique, linéarisation	11
6.2.3	Réarrangement des blocs de base	13

6.1 Le code intermédiaire

6.1.1 Préliminaires

Structure de la traduction

La traduction d'un programme a la structure suivante :



où chaque fragment est soit une chaîne de caractères, soit le code associé à *une* fonction (le « main » étant une fonction comme les autres).

```
type 'a proc_desc = {  
  entry: lbl;    (** Entry point, will the prologue's lbl *)  
  numargs: int;  (** Number of argument *)  
  numvars: int;  (** Number of words in the frame for escaping variables *)  
  code: 'a;      (** Procedure's code.  
                  It would first be Ir.ext and then Lin.funbody *)  
}  
  
type 'code prog = {
```

```

    strings: (lbl * string) list; (** List of strings to be declared in .data *)
    procs: 'code proc_desc list;  (** List of procedure *)
}

```

Une parenthèse : comment traiter « main »

On peut aisément rendre uniforme le traitement du corps du programme (le « main »), avec le traitement de toute autre fonction ordinaire, avec une petite astuce :

- on parse le programme P et l'on produit son AST a
- on produit un nouvel AST a' qui est équivalent à celui qui serait produit par

```

var main () = (P; 0) in
main ()

```

- on continue la compilation sur a', qui ne contient plus, dans le corps, aucune déclaration

Dans le compilateur du projet, les lignes suivantes font cette transformation :

```

let mainwrapper exp =
  LetFunExp
    ([{fun_id = "main";
      fun_params = [];
      fun_res = Some "int";
      fun_body = SeqExp(exp, IntExp(0))}],
     Apply ({f_name = "main"; f_level = None}, []))

```

6.1.2 Description du code intermédiaire

Code Intermédiaire arborescent

Nous introduisons maintenant le premier langage intermédiaire vers lequel vous allons *traduire* notre langage source. Il s'agit encore d'une représentation *arborescente*, mais dans laquelle les instructions disponibles sont beaucoup plus proches des instructions machines ; on retrouve en effet :

- des étiquettes, et des sauts (conditionnels ou pas) à des étiquettes
- des accès mémoire
- des déplacements de données
- des opérations de comparaison
- des opérations arithmétiques
- l'instruction CALL

AST du Code Intermédiaire arborescent

Contenu du fichier `ir.mli` du projet :

```

(** Ir : Intermediate Representation *)

type binop = PLUS | MINUS | MUL | DIV
           | AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR

type relop = EQ | NE | LT | GT | LE | GE | ULT | ULE | UGT | UGE

```

```
type stm =  
  | LABEL
```

6.1.3 Traduction en code intermédiaire

La traduction T vers le code intermédiaire est longue, mais sans surprise. Nous allons voir quelques cas, le reste étant laissé pour le projet. Nous remarquons que la traduction sera effectuée en ayant accès pour toute variable simple aux attributs *level* et *offset*, et pour toute fonction à l'attribut *level*, calculés comme expliqué avant.

On supposera que chaque variable occupe exactement un mot mémoire (de taille W octets, selon la machine). Pour les types complexes, ce mot mémoire contiendra *un pointeur* vers la structure allouée dans le tas et pas dans la pile.

Ces conventions nous permettent de nous passer de la distinction habituelle entre valeurs gauches et valeurs droites, qu'il faut quand même rappeler brièvement ici.

L-Values (valeurs gauches) et R-values (valeurs droites)

On distingue dans la littérature (et notamment dans les manuels C) deux types de valeurs :

L-values les valeurs qui peuvent apparaître à *gauche* d'une affectation (les valeurs qui désignent des cases mémoire dans lesquelles on peut écrire). En CTigre, c'est le cas des

variables simples comme x ;

champs d'enregistrements comme $a.nom$;

éléments d'un tableau comme $a[3+5]$.

R-values les valeurs qui peuvent apparaître à *droite* d'une affectation (les expressions qui ont une valeur que l'on peut écrire dans des cases mémoire). En CTigre, c'est le cas de tous les L-values, mais aussi d'expressions qui ne sont pas L-values :

expressions arithmétiques comme $1+x-32$;

fonctions qui retournent des types de base comme $\text{succ}(1)$.

Comme les L-values sont le plus souvent aussi des R-values, il est nécessaire de regarder le contexte pour savoir s'il faut produire du code qui *lit* une valeur depuis la case mémoire, ou du code qui *écrit* une valeur dans la case mémoire.

Dans le cas de CTigre, nous pouvons éviter cette analyse grâce à *deux* hypothèses simples :

- toutes les variables prennent la même place (cela se fait en forçant les variables de type tableaux et enregistrement à contenir un pointeur vers la mémoire plutôt que le tableau ou enregistrement tout entier) ;
- la construction MEM du langage intermédiaire ne préjuge pas de la lecture ou écriture, qui est décidée par le contexte.

Dans ce qui suit, la traduction d'une variable simple ou composée sera donc toujours la même sans se soucier de savoir si elle est en position droite ou gauche.

Traduction : VarExp

Le cas des variables : La traduction de l'accès à une SimpleVar de *level* l et *offset* o sera la suivante :

- dans la fonction f qui la déclare ($l = \text{level}(f)$) :

MEM(BINOP(MINUS, FP, WORDMUL(CONST o)))

- dans une fonction g englobée par f ($l = \text{level}(f) < \text{level}(g)$), on suit le lien statique :

MEM(BINOP(MINUS, (MEM(... MEM(FP) ...), WORDMUL(CONST o))))
level(g)-1 fois

Traduction : éléments d'un vecteur

La traduction de l'accès à un élément d'un vecteur, $e[e1]$ sera traitée comme suit :

```
MEM(BINOP(PLUS, MEM(T(e)), WORDMUL(T(e1))))
```

où $T(e)$, $T(e1)$ sont les traductions de e et $e1$.

Traduction : boucle while

La traduction de while b do c done sera

```
[LABEL(test);  
  CJUMP(EQ, T(b), CONST(1), cont, done);  
  LABEL(cont);  
  T(c);  
  JUMP(test);  
  LABEL(done)]
```

La traduction de while b do c done devient plus claire si on la visualise de la façon suivante.

```
test:      CJUMP(EQ, T(b), CONST(1), cont, done)  
cont:     T(c)  
          JUMP test  
done:
```

Traduction : boucle for

Traduire en boucle while.¹.

Traduction : Appel d'une fonction

La traduction d'un appel de fonction $f(a1, \dots, an)$ est immédiate

```
CALL(lf, [ls, T(a1), ..., T(an)])
```

Ne pas oublier le lien statique ls qui est ajouté en paramètre.

Rappel : Pour calculer ls il vous faut le *level* de f (connu dans l'environnement) et celui de la fonction g qui appelle f (facile à connaître, parce que vous êtes en train de traduire g en ce moment). Exemple : $MEM(MEM(FP))$ si la différence de niveau entre l'appelant et l'appelé est 1.

Traduction des chaînes de caractères

Une chaîne de caractères est normalement traduite en langage d'assemblage par une étiquette repérant l'adresse d'une directive spéciale suivie du texte de la chaîne, que l'assembleur traduira ensuite en une séquence de mots, terminés ou pas par un octet à zéro, dans une zone mémoire spéciale (généralement appelée section « data »).

Dans le cas de l'émulateur MARS, vous trouvez par exemple des fragments de code comme

```
.data  
bonjour:  
  .asciiz "Bonjour tout le monde!\n"
```

¹ Il faudra faire attention à l'ordre d'évaluation des bornes qui doit être cohérent avec le calcul des levels et offsets (dans le cas où l'on réutilise plusieurs fois le même offset)

Les opérations utilisant cette chaîne de caractères feront référence à l'étiquette assembleur `bonjour`.

Traduction : déclaration de variable

variable La déclaration d'une variable `var a:=e in ...` produira une expression qui initialise cette variable (dans le bloc d'activation courant, à *offset* connu) avec $T(e)$.

Traduction : déclaration de fonction

fonction La déclaration d'une fonction produira :

- une étiquette unique lf , associée à la séquence d'instructions de la fonction
- les informations pour construire le prologue et l'épilogue (place à réserver pour les variables locales de la fonction)
- le corps de la fonction

Important : la traduction d'une fonction produit un *nouveau* `Ir.stms`

6.1.4 EXP et ESEQ

Traduction : utilisation de EXP et ESEQ

Dans notre langage intermédiaire, une *expression* (type `exp`) est un arbre d'instructions qui retourne une valeur, alors qu'une *instruction* (« statement », type `stm`) est un arbre d'instructions ne rendant pas de valeur. Mais il y a des situations, pendant la traduction, qui nous obligent à utiliser l'un pour traduire l'autre et vice-versa.

Le code intermédiaire dispose des deux constructeurs de séquençement :

`type exp = ... | ESEQ of stm * exp | ...`

Voyons quelques exemples d'utilisation de EXP et ESEQ :

EXP on utilise cette conversion pour les « programmes » constitués d'une expression. C'est le cas, par exemple, du programme CTigre

3+4

qui évalue l'expression mais ne fait rien de son résultat.

ESEQ on utilise cette conversion quand, pour calculer une valeur, on a besoin d'effectuer d'abord une série d'instructions. C'est le cas de

- la traduction des opérateurs booléens
- l'initialisation de variables de type structuré (tableaux, enregistrements)

ESEQ et opérateurs booléens

En langage machine, les opérateurs de comparaison comme \geq , \leq , etc. sont souvent disponibles *seulement* pour permettre un branchement lors d'un saut, donc on ne peut pas traduire :

`a+b <= 32`

par une expression

`BINOP(LT, .. , ..)`

Ce fait est bien mis en évidence dans notre langage intermédiaire par le fait que les opérateurs de relation comme LT, GT, EQ, NE etc. sont utilisables seulement dans l’instruction intermédiaire CJUMP(*op*, *e1*, *e2*, *t*, *f*), qui saute à l’étiquette *t* ou *f* selon que l’opération *op* sur *e1* et *e2* a résultat vrai ou faux.

Cela nous conduit à une traduction qui nécessite l’utilisation de ESEQ.

Pour traduire

$$exp_1 \text{ op } exp_2$$

avec *op* opérateur de comparaison, nous allons produire d’abord la séquence d’instructions

```
CJUMP(op, T(exp1), T(exp2), 10, 11)
LABEL 10
MOVE(TEMP t, CONST 1)
JUMP 12
LABEL 11
MOVE(TEMP t, CONST 0)
LABEL 12
```

avec 10, 11, 12 des nouvelles étiquettes et *t* un nouveau nom de « registre temporaire ». À la fin de cette suite d’instructions, nous retrouvons dans le temporaire *t* la valeur correspondante à la comparaison (1 pour vrai et 0 pour faux). On peut alors, à l’aide de ESEQ, récupérer la valeur de *t*.

Très précisément, la traduction sera

```
ESEQ(
  [CJUMP(op, T(exp1), T(exp2), L0, L1);
   LABEL L0;
   MOVE(TEMP t, CONST 1);
   JUMP L2;
   LABEL L1;
   MOVE(TEMP t, CONST 0);
   LABEL L2],
  TEMP t)
```

Données allouées sur le tas

Le tas est une zone de mémoire distincte de la pile affectée au processus par le système. On y stocke les données allouées dynamiquement. C’est au programme de gérer cet espace mémoire comme il l’entend. La bibliothèque standard C fournit les fonctions `malloc` et `free` qui permettent d’allouer et libérer de l’espace mémoire dans cette zone. Des langages plus modernes (Java, OCaml) font cela automatiquement (utilisation d’un *garbage collector* pour libérer l’espace).

MARS implémente un appel système (`sbrk`), qui permet d’augmenter la taille de la zone mémoire du tas. Il faut gérer nous-même l’allocation (et, dans l’idéal la libération) de mémoire dans cette zone.

Nous supposons que nous disposons d’une fonction externe (du système), qui sait allouer de la mémoire dans le tas. Par simplicité, ici on utilisera le nom `malloc` et on supposera que l’on puisse appeler cette fonction comme une fonction CTigre quelconque. Cette fonction est fournie dans le fichier `runtime.s` du projet.

ESEQ et types structurés

Supposons d'avoir à traduire un fragment comme celui-ci

```
type arrtype = array of int in  
var arr1:arrtype := arrtype [10] of 0
```

Au moment de la traduction de la variable `arr1`, nous devons produire d'abord une suite d'instructions qui allouera dans le tas l'espace mémoire nécessaire pour contenir le tableau de 10 entiers, puis initialiser chaque case à 0 et ensuite retourner comme traduction de la variable un « pointeur » sur le début de ce tableau, c'est à dire l'adresse de la première case mémoire du tableau.

Nous avons besoin d'utiliser ESEQ, mais aussi de faire appel à `malloc`.

La traduction de

```
type arrtype = array of int in  
var arr1:arrtype := arrtype [10] of 0
```

sera donc la suite d'instructions

```
ESEQ(  
[MOVE(TEMP t, CALL(malloc, [1s, (CONST (W * 10))]))];  
MOVE(MEM(BINOP(PLUS, TEMP t, (CONST (W * 0))), CONST 0));  
.  
.  
.  
MOVE(MEM(BINOP(PLUS, TEMP t, (CONST (W * 9))), CONST 0))],  
TEMP t)
```

Remarque : Dans ce code, `W` est la taille du mot en octets (sur les machines modernes, c'est au moins 2, souvent 4 et de plus en plus 8). Aussi, les multiplications dans les `CONST` ici sont faites à la compilation, en connaissant `W`, cela ne fait pas partie de la syntaxe du code intermédiaire !

Remarque : Cette traduction ne fonctionne que si la taille du tableau est une constante... Si elle est une expression calculée à l'exécution, on doit traduire différemment, en produisant du code assembleur qui va exécuter une boucle d'initialisation (et alors, les calculs sur les `W` seront fait à l'exécution avec `WORDMUL` !)

Les enregistrements seront aussi placés sur le tas.

Chaque champ sera codé sur exactement un mot. On aura besoin d'une table des symboles qui associe à chaque nom de champ son décalage (numéro de champ).

6.1.5 Structure de la fonction de traduction

On a besoin de deux tables de symboles :

- Une contenant les labels (et en fait toute l'information sur le bloc) associés à chaque nom de fonction (les labels doivent être uniques) ;
- une contenant les décalages des champs des enregistrements.

```
transexp :  
Label.lbl Symbol.table ->  
int Symbol.table -> int -> Ast.attexp -> Ir.exp
```

CTigre autorise la définition de structures de données récursives ou mutuellement récursives.
Exemples :


```

type intlist = {hd: int, tl: intlist} in

var lis:intlist := intlist { hd=0, tl= lis } in

var l := intlist { hd = 1, tl = l' }
and l' := intlist { hd = 2, tl = l } in ...

```

Pour traiter ces cas, il faut d'abord faire l'allocation de toutes les structures du LetVarExp, et ensuite le remplissage des champs.

Cela n'est possible que si le bloc est construit tout de suite :

Objective Caml version 3.10.1

```

# let rec l = 1::l;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
# let f l = 1::l;;
val f : int list -> int list = <fun>
# let rec l = f l;;
This kind of expression is not allowed as right-hand side of 'let rec'

```

Pour compiler des fonctions mutuellement récursives, on a besoin de connaître le label associé à chacune des fonctions avant de générer le code de la première fonction.

Pour cela, on fait la compilation d'un LetFunExp en deux passes : la première pour remplir la table des symboles des frames, la deuxième pour générer le code.

6.2 Transformations du code intermédiaire

6.2.1 Une pause de réflexion

Remarques sur le code intermédiaire

1. Le code est linéaire pour les instructions
2. Branchements explicites, vers des étiquettes (on vise un processeur qui exécute les instructions séquentiellement, avec des branchements)
3. Le code est arborescent pour les expressions (la traduction des expressions dépend beaucoup du processeur cible)
4. Instruction d'appel de fonctions (la traduction des appels de fonctions dépend beaucoup du processeur cible)
5. Infinité de registres (nous ne savons pas combien le processeur cible en a)

Rendre le code intermédiaire le plus indépendant possible du processeur

Avant d'aller plus loin, réfléchissons à ce que l'on est en train de faire : nous avons pris un programme en entrée (sous forme d'arbre de syntaxe abstraite typé et annoté avec les attributs *level* et *offset*), et nous le traduisons maintenant vers du code intermédiaire.

- le numéro du registre qui contient le FP (30 sur MIPS) ou le SP,
- la taille d'un mot en mémoire (4 sur MIPS),
- les conventions d'émission des chaînes de caractères, d'appel des fonctions de bibliothèques, etc.

Il serait bien de garder ces dépendances groupées dans un module bien identifié.

Structure du résultat

De plus, le résultat de cette traduction n'est pas un seul `Ir.stms`, mais plutôt *une liste de fragments* de code distincts :

- un fragment pour toute définition de fonction (ce fragment contient le prologue, le corps et l'épilogue de la fonction)
- un fragment pour le « main », le corps du programme principal (qui est juste une fonction englobant toutes les autres)
- un fragment pour toute chaîne de caractère dans le programme (celui-ci, on ne le traduira pas en Code Intermédiaire, mais on le gardera dans la liste pour la phase d'émission de code assembleur)

Se rapprocher encore de l'assembleur

Enfin, le code intermédiaire que nous produisons n'est pas, tel quel, prêt à être converti vers du langage machine.

- On souhaite pouvoir produire le code pour évaluer une expression dans n'importe quel ordre (notamment pour minimiser le nombre de registres nécessaires pour l'évaluer). Mais certaines configurations d'un arbre de code intermédiaire font en sorte que l'ordre d'évaluation devient *significatif* en général :
 - Les noeuds ESEQ dans une `Ir.exp` (ils peuvent produire des effets de bord)
 - Des noeuds CALL dans une `Ir.exp` (idem)
- Dans les compilateurs modernes, on renvoie souvent le (pointeur sur le) résultat dans un même registre ; il est donc mieux de sauvegarder immédiatement le résultat d'un CALL dans un autre registre (sauf si ce résultat est ignoré, pour les procédures, quand CALL est fils de EXP).
- Les instructions CJUMP du code intermédiaire peuvent sauter à deux adresses différentes, alors que les sauts conditionnels en langage machine ont toujours l'instruction suivante comme étiquette de saut en cas de échec.

Dans la suite de cette section, nous allons faire deux passes de transformation sur le code intermédiaire pour le rapprocher encore de l'assembleur : la linéarisation et l'optimisation du flot de contrôle.

Le nouveau code intermédiaire (module Lin)

```
type exp =
| LBINOP of Ir.binop * exp * exp
| LMEM of exp
| LWORDMUL of exp
| LFP
```

```

| LTEMP of temp
| LCONST of int
| LSTR of lbl

```

```

type stm =
| LLABEL of lbl
| LJUMP of lbl
| LCJUMP of Ir.relop * exp * exp * lbl * lbl
| LMOVETEMP of temp * exp
| LMOVEMEM of exp * exp
| LMOVETEMPCALL of temp * lbl * exp list
| LEXPCALL of lbl * exp list

```

```

type stms = stm list

```

Utiliser ce type permet de garantir *par typage statique* que notre transformation est correcte !

6.2.2 Mise en forme canonique, linéarisation

On peut toujours convertir du code intermédiaire dans du code intermédiaire qui n'a pas les inconvénients cités ci-dessus.

Definition 1 (Arbre canonique). Un arbre *canonique* est un arbre de code intermédiaire tel que

- il n'y a pas de noeuds ESEQ
- le père des noeuds CALL est toujours soit un EXP soit un MOVE(TEMP t , $_$).

On va définir une première transformation, qui transforme tout arbre en arbre canonique.

Mise en forme canonique

Cette transformation est basée sur une idée très simple :

- on remplace tout appel de fonction CALL qui n'est pas déjà le fils d'un EXP ou un MOVE(TEMP t , $_$) par la séquence équivalente

$$\text{ESEQ}([\text{MOVE}(\text{TEMP } t, \text{CALL}(f, el))], \text{TEMP } t)$$

- on fait remonter tout ESEQ qui se trouve à l'intérieur d'une expression, jusqu'en haut, où le ESEQ peut disparaître ; pour cela, on peut être amené à introduire des nouveaux temporaires pour les expressions qui ne commutent pas avec les commandes que l'on fait remonter

Remontée des ESEQ

Voyons un exemple : le fragment

```

BINOP(PLUS,
      ESEQ(s, e1),
      e2)

```

peut être remplacé par le fragment équivalent

```
ESEQ(s,
  BINOP(PLUS,
    e1,
    e2)
)
```

Par contre, le fragment

```
BINOP(PLUS,
  e1,
  ESEQ(s, e2)
)
```

peut être remplacé par le fragment

```
ESEQ(s,
  BINOP(PLUS,
    e1,
    e2)
)
```

seulement si s et $e1$ commutent (i.e. si le fait d'exécuter s puis $e1$ ou $e1$ puis s ne change pas la sémantique du programme).

Si l'on ne sait pas prouver la commutation, on doit introduire un nouveau temporaire t et utiliser le fragment (toujours équivalent) :

```
ESEQ(MOVE(TEMP t, e1),
  ESEQ(s,
    BINOP(PLUS,
      TEMP t,
      e2)
    )
  )
```

Pour toute possible configuration contenant un ESEQ à l'intérieur, on peut donner un fragment de code équivalent où l'ESEQ a soit disparu soit été remonté vers la racine. Voici quelques autres exemples :

Le fragment	est remplacé par
MEM(ESEQ(s,e))	ESEQ(s, MEM(e))
JUMP(ESEQ(s,e))	[s ; JUMP(e)]
CJUMP(op, ESEQ(s,e1), e2, t, f)	[s ; CJUMP(op, e1, e2, t, f)]
MOVE(TEMP t, ESEQ(s,e))	[s ; MOVE(TEMP t, e)]

Si s et $e1$ commutent,

Le fragment	est remplacé par
BINOP(op, e1, ESEQ(s, e))	ESEQ(s, BINOP(op, e1, e))
CJUMP(op, e1, ESEQ(s, e), t, f)	[s ; CJUMP(s, e1, e, t, f)]

Et s'ils ne commutent pas :

Le fragment	est remplacé par
BINOP(op, e1, ESEQ(s, e))	ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e)))
CJUMP(op, e1, ESEQ(s,e), t, f)	[MOVE(TEMP t, e1) ; s ; CJUMP(op, TEMP t, e, t, f)]

Exercice : complétez l'ensemble des règles.

Une fois les ESEQ arrivés en tête, on peut les supprimer, sans problème :

$\text{EXP}(\text{ESEQ}(s, e))$

devient

$[s; \text{EXP}(e)]$

et en général,

$\text{EXP}(\text{ESEQ}(s_1, \text{ESEQ}(s_2, \dots \text{ESEQ}(s_n, e)) \dots))$

devient

$[s_1; s_2; \dots s_n; \text{EXP}(e)]$

Réécriture

Un ensemble de règles comme celui que l'on vient de voir donne lieu, une fois fermé par contexte, à ce que l'on appelle un *système de réécriture*, qui est étudié en informatique théorique, où l'on donne des méthodes pour montrer qu'un ensemble de règle *termine* (i.e. le programme qui opère la transformation finit toujours) et est *confluent* (i.e. peu importe l'ordre d'application des règles, l'arbre canonique est toujours le même). Vous pourriez en savoir plus, par exemple, en suivant les cours du Master Recherche à Paris VII.

Remontée des ESEQ : alternative

On peut aussi remarquer que l'application de ces règles revient à extraire de tout arbre d'expression e les instructions s_1, \dots, s_n produisant des effets de bord et encapsulées dans des $\text{ESEQ}(s_i, e_j)$. Cela produira une séquence de Ir.stm , suivie d'une Ir.exp sans effets de bord.

Au cours de cette extraction, il se peut que l'on rencontre, comme on l'a vu, des expressions qui ne commutent pas avec une instruction que l'on veut remonter, et alors on passe l'expression dans la partie instruction en introduisant un nouveau temporaire qui remplace l'expression.

6.2.3 Réarrangement des blocs de base

Une fois notre arbre mis en forme canonique, nous nous retrouvons avec une liste d'instructions et expressions sans effets de bord. Il nous reste à traiter la deuxième différence entre code intermédiaire et assembleur : les *deux* étiquettes de LCJUMP.

Nous souhaitons réordonner la liste d'instructions pour essayer de faire en sorte que la deuxième étiquette d'un LCJUMP apparaisse juste après le LCJUMP dans la liste (quand cela ne sera pas possible, on introduira de nouvelles étiquettes).

Blocs de base

Pour pouvoir réordonner notre code sans changer la signification du programme, nous devons d'abord le découper en *blocs de base*.

Definition 2 (bloc de base). Un *bloc de base* est une séquence d'instructions ayant les propriétés suivantes :

- la première instruction est un LLABEL
- la dernière instruction est un LJUMP ou un LCJUMP
- il n’y a pas d’autres LLABEL, LJUMP ou LCJUMP dans le bloc

Pour produire une liste de blocs de base, on parcourt la liste d’instructions, on crée un nouveau bloc dès que l’on rencontre un LLABEL, et on le termine dès que l’on rencontre un LJUMP ou LCJUMP.

Si on se trouve avec un bloc sans LLABEL initial, on crée une nouvelle étiquette que l’on met en tête de ce bloc.

Si on trouve un bloc qui ne se termine pas avec un LJUMP ou LCJUMP, on rajoute un LJUMP vers l’étiquette du nouveau bloc.

Algorithme BasicBlocks

Voilà le code OCaml

```
let basicBlocks stms =
  let donel = Temp.new_label() in
  let rec blocks = function
    ((LLABEL _ as head) :: tail, blist) ->
      let rec next (stms, thisblock) = match stms with
        | (LJUMP _ as s)::rest -> endblock(rest, s::thisblock)
        | (LCJUMP _ as s)::rest -> endblock(rest, s::thisblock)
        | LLABEL lab :: _ -> next(JUMP lab :: stms, thisblock)
        | s::rest -> next(rest, s::thisblock)
        | [] -> next([LJUMP donel], thisblock)
      and endblock(stms, thisblock) =
        blocks(stms, List.rev thisblock :: blist)
      in next(tail, [head])
  | ([], blist) -> List.rev blist
  | (stms, blist) -> blocks(T.LLABEL(Temp.new_label())::stms, blist)
in (blocks(stms, []), donel)
```

Traces

Une fois que nous avons notre liste de blocs de base, on peut les réarranger dans n’importe quel ordre, parce que les sauts et les étiquettes garantissent que le flot du contrôle sera toujours le bon. On peut profiter de cette propriété pour chercher un ordre de rearrangement qui résout notre problème de doubles étiquettes.

Definition 3. Une *trace* est une séquence d’instructions (y compris sauts et sauts conditionnels) qui peuvent être exécutés consécutivement lors de l’exécution du programme.

Un programme possède de multiples traces, qui s’intersectent : nous cherchons un ensemble de traces qui couvre sans répétition tout le programme. On veut maximiser le nombre de LJUMP ou LCJUMP suivi par une des leurs étiquettes, pour optimiser la suite.

Traces : exemple

Au tableau

Trouver les traces : un algorithme simple

Pour trouver cet ensemble de traces, en commençant avec une liste L de blocs de bases, on peut utiliser ce simple algorithme :

```
tant que L n'est pas vide faire
  initialiser une nouvelle trace vide T
  enlever l'élément de tête a de L
  tant que a n'est pas marqué faire
    marquer a
    ajouter a à la trace T
    examiner les successeurs de a
    si il y a un successeur c non marqué
      alors a <- c
  fin faire
  clore la trace T
fin faire
```

Étape finale

Une fois obtenu l'ensemble de traces, nous pouvons le transformer à nouveau en une liste d'instructions, et ensuite examiner cette liste pour quelques opérations de finalisation :

- tout LCJUMP qui est suivi par l'étiquette correspondante à la branche false est déjà correcte
- pour tout LCJUMP qui est suivi par l'étiquette correspondante à la branche true, on échange ses étiquettes true et false et on remplace le test par son complémentaire (EQ par NE, LT par GE etc.)
- pour tout LCJUMP(op, e1, e2, t, f) qui n'est suivi par aucune des ses deux étiquettes t et f, on invente une nouvelle étiquette f' et on le remplace par la séquence :

```
LCJUMP (op, e1, e2, t, f')
LLABEL f'
LJUMP (f)
```

- tout LJUMP suivi immédiatement par son étiquette est éliminé

Une fois cela fait, tout LCJUMP est suivi par son étiquette false et on n'a pas de LJUMP immédiatement suivis par leur étiquette.

Remarque : un compilateur industriel chercherait à trouver un ensemble de traces qui minimise les sauts dans les sections critiques du code (boucle, etc.).

Conclusions

Nous avons, à la fin de ce cours, pu produire une séquence d'instructions en code intermédiaire qui n'a pas d'effets de bord dans les expressions et qui est assez proche de l'assembleur pour pouvoir être converti aisément dans une représentation intermédiaire de plus bas niveau (une sorte de assembleur abstrait), comme nous verrons dans le chapitre suivant.