Description de quelques compilateurs

Luc Maranget

2003

Source Pseudo-Pascal

```
function power(var x : integer, var n : integer) : integer ;
var r : integer ;
var t : integer ;
begin
    t := x ; r := 1 ;
    while n > 0 do begin
        if n <> 2*(n/2) then
            r := r * t
        else begin
        end ;
        n := n / 2 ;
        t := t*t
    end ;
    power := r
end ;
```

Assembleur produit par zyva

```
#FUNSTART power
    power_f=0
power:
    li   $v0, 1
    b    L13
L12:
    sra  $a2, $a1, 1
    sll  $a2, $a2, 1
    beq  $a1, $a2, L17
L15:
    mul  $v0, $v0, $a0
L17:
    sra  $a1, $a1, 1
    mul  $a0, $a0, $a0
L13:
    bgt  $a1, $zero, L12
L14:
power_end:
    j    $ra
```

Source C

```
int power(int x, int n) {
  int r=1 ;
  int t=x ;

  for ( ; n > 0 ; n /= 2) {
    if ((n % 2) != 0)
      r *= t ;
    t = t * t ;
  }
  return r ;
}
```

Assembleur (Mips) produit par gcc

Prologue

```
power:
.LFB1:
        .frame  $sp,32,$31
        subu    $sp,$sp,32
.LCFI0:
        sd      $28,16($sp)
.LCFI1:
        .set    noat
        lui     $1,%hi(%neg(%gp_rel(power)))
        addiu   $1,$1,%lo(%neg(%gp_rel(power)))
        daddu   $gp,$1,$25
        .set    at
```

Épilogue

```
.L18:
        move    $2,$6
        ld      $28,16($sp)
        .set    noreorder
        j       $31
        addu    $sp,$sp,32
        .set    reorder
```

Assembleur (Mips) produit par gcc

```
        .set    noreorder
        blez    $5,.L18
        li      $6,1
        .set    reorder
        srl     $7,$5,31
        .align  3
.L19:
        addu    $3,$5,$7
        andi    $7,$5,0x1
        .set    noreorder
        beq     $7,$0,.L6
        sra     $5,$3,1
        .set    reorder
        mult    $6,$4
        mflo    $6
.L6:
        mult    $4,$4
        mflo    $4
        .set    noreorder
        bgtz    $5,.L19
        srl     $7,$5,31
        .set    reorder
.L18:
```

Retrouver les registres, delay-slots, arithmétique signée.

Assembleur (Pentium) produit par gcc

```
power:
        pushl %ebp
        movl %esp,%ebp
        pushl %ebx
        movl 12(%ebp),%edx
        movl $1,%ebx
        movl 8(%ebp),%ecx
        testl %edx,%edx
        jle .L18
        .p2align 4,,7
.L20:
        testb $1,%dl
        je .L21
        imull %ecx,%ebx
.L21:
        imull %ecx,%ecx
        movl %edx,%eax
        shrl $31,%eax
        addl %eax,%edx
        sarl $1,%edx
        testl %edx,%edx
        jg .L20
.L18:
        movl %ebx,%eax
        movl -4(%ebp),%ebx
        leave
        ret
```

Frame pointer, excellente sélection, allignements. Callee-save ?.

Source Caml

```
let rec power n r t = match n with
| 0 -> r
| _ ->
    if n mod 2 <> 0 then power (n/2) (r*t) (t*t)
    else power (n/2) r (t*t)
```

Prologue

```
Poweriter_power_49:
        subu   $sp, $sp, 16
        sw     $31, 12($sp)
        sw     $gp, 8($sp)
        lui    $24, %hi(%neg(%gp_rel(Poweriter_power_49)))
        addiu  $24, $24, %lo(%neg(%gp_rel(Poweriter_power_49)))
        daddu  $gp, $25, $24
$102:
```

Épilogue

```
$100:
        move   $2, $9
        lw     $gp, 8($sp)
        addu   $sp, $sp, 16
        .livereg 0x20000302, 0x00000000
        j      $31
```

Le code assembleur produit par `ocamlopt`

```
$102:
        beq     $8, 1, $100
        sra     $7, $8, 1
        rem     $6, $7, 2
        sll     $5, $6, 1
        addu    $4, $5, 1
        beq     $4, 1, $101
        sra     $15, $10, 1
        addu    $14, $10, -1
        mul     $13, $14, $15
        addu    $13, $13, 1
        sra     $12, $10, 1
        addu    $11, $9, -1
        mul     $9, $11, $12
        addu    $9, $9, 1
        sra     $7, $8, 1
        div     $6, $7, 2
        sll     $5, $6, 1
        addu    $8, $5, 1
        move    $10, $13
        b       $102
$101:
        sra     $3, $10, 1
        addu    $2, $10, -1
        mul     $21, $2, $3
        addu    $10, $21, 1
        sra     $19, $8, 1
        div     $18, $19, 2
        sll     $17, $18, 1
        addu    $8, $17, 1
        b       $102
$100:
```

Arithmétique « $2n + 1$ », faiblesse sur le modulo/division (assembleur ?), suppression de la récursion terminale. Sauvegarde (inutile) de `ra`.

Le code Pentium produit par `ocamlopt`

```
Poweriter_power_49:
.L102:
        movl    %eax, %esi
        cmpl    $1, %esi
        je      .L100
        movl    %esi, %edx
        sarl    $1, %edx
        movl    %edx, %eax
        testl   %eax, %eax
        jge     .L103
        addl    $1, %eax
.L103:  andl    $-2, %eax
        subl    %eax, %edx
        lea     1(, %edx, 2), %eax
        cmpl    $1, %eax
        je      .L101
        movl    %ecx, %eax
        sarl    $1, %eax
        movl    %ecx, %edx
        decl    %edx
        imull   %eax, %edx
        incl    %edx
        sarl    $1, %ecx
        decl    %ebx
        imull   %ecx, %ebx
        incl    %ebx
        sarl    $1, %esi
        testl   %esi, %esi
        jge     .L104
        addl    $1, %esi
.L104:  sarl    $1, %esi
        lea     1(, %esi, 2), %eax
        movl    %edx, %ecx
        jmp     .L102
```

Le code Pentium produit par ocamlopt

```
        .align  16
.L101:
        movl    %ecx, %eax
        sarl    $1, %eax
        decl    %ecx
        imull   %eax, %ecx
        incl    %ecx
        sarl    $1, %esi
        testl   %esi, %esi
        jge     .L105
        addl    $1, %esi
.L105:  sarl    $1, %esi
        lea     1(, %esi, 2), %eax
        jmp     .L102
        .align  16
.L100:
        movl    %ebx, %eax
        ret
```

Noter, les conventions d'appel (meilleures que gcc ici), la sélection (moins bonne).

Comment en arrive-t-on là ?

Code intermédiaire

```
function power
args = $t108 $t109
result = $t107
 Seq
  (set $t110 $t108)
  (set $t111 1)
  Jump L13
 L12:
  Cjump L15 L16
   (<> $t109 (* 2 (/ $t109 2)))
 L16:
  Jump L17
 L15:
  (set $t111 (* $t111 $t110))
 L17:
  (set $t109 (/ $t109 2))
  (set $t110 (* $t110 $t110))
 L13:
  Cjump L12 L14 (> $t109 0)
 L14:
  (set $t107 $t111)
```

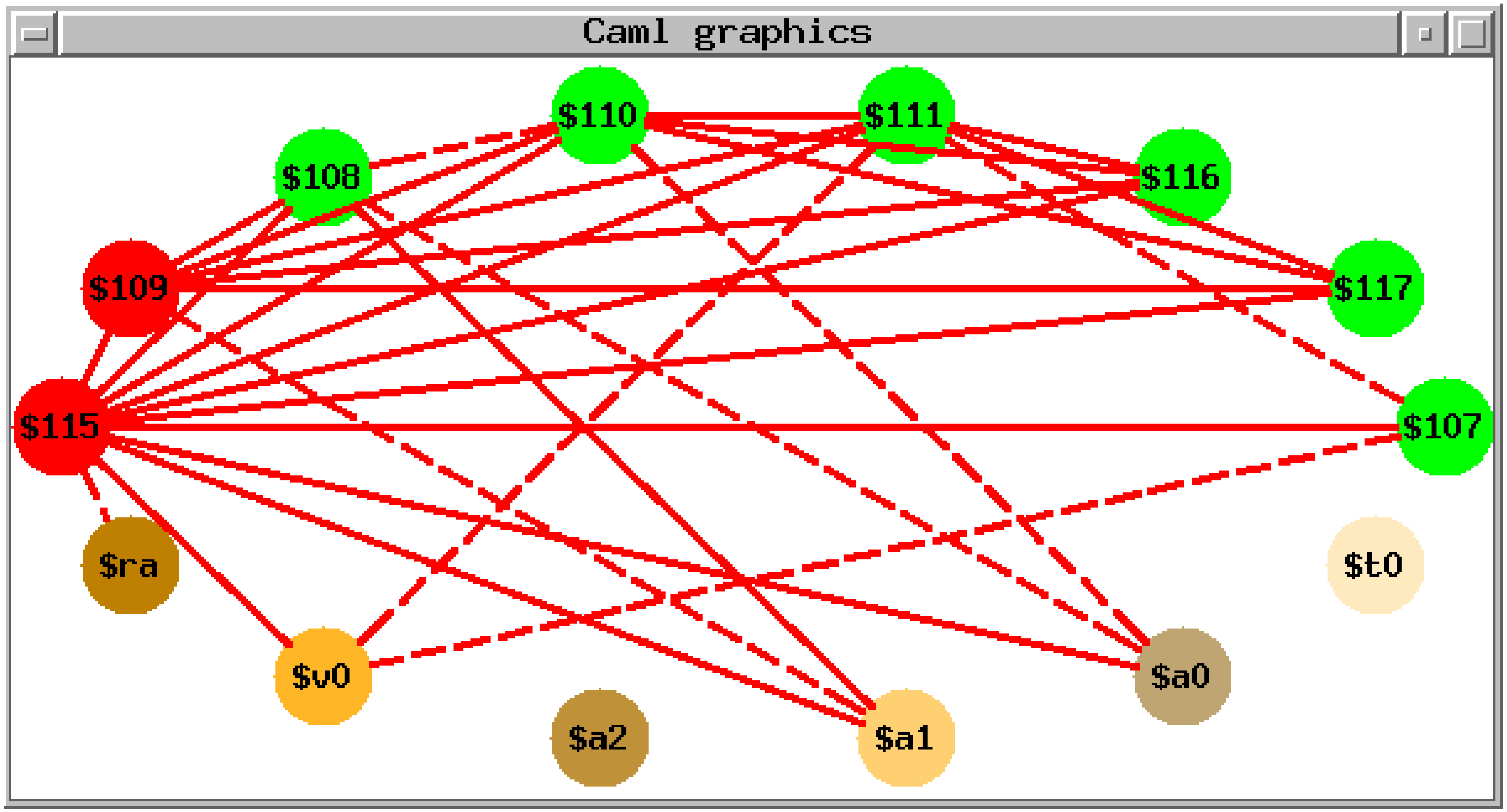
Code intermédiaire cannonique

```
function power
args = $t108 $t109
result = $t107
  (set $t110 $t108)
  (set $t111 1)
  Jump L13
L12:
  Cjump L17 L15
    (= $t109 (* 2 (/ $t109 2)))
L15:
  (set $t111 (* $t111 $t110))
L17:
  (set $t109 (/ $t109 2))
  (set $t110 (* $t110 $t110))
L13:
  Cjump L12 L14 (> $t109 0)
L14:
  (set $t107 $t111)
```

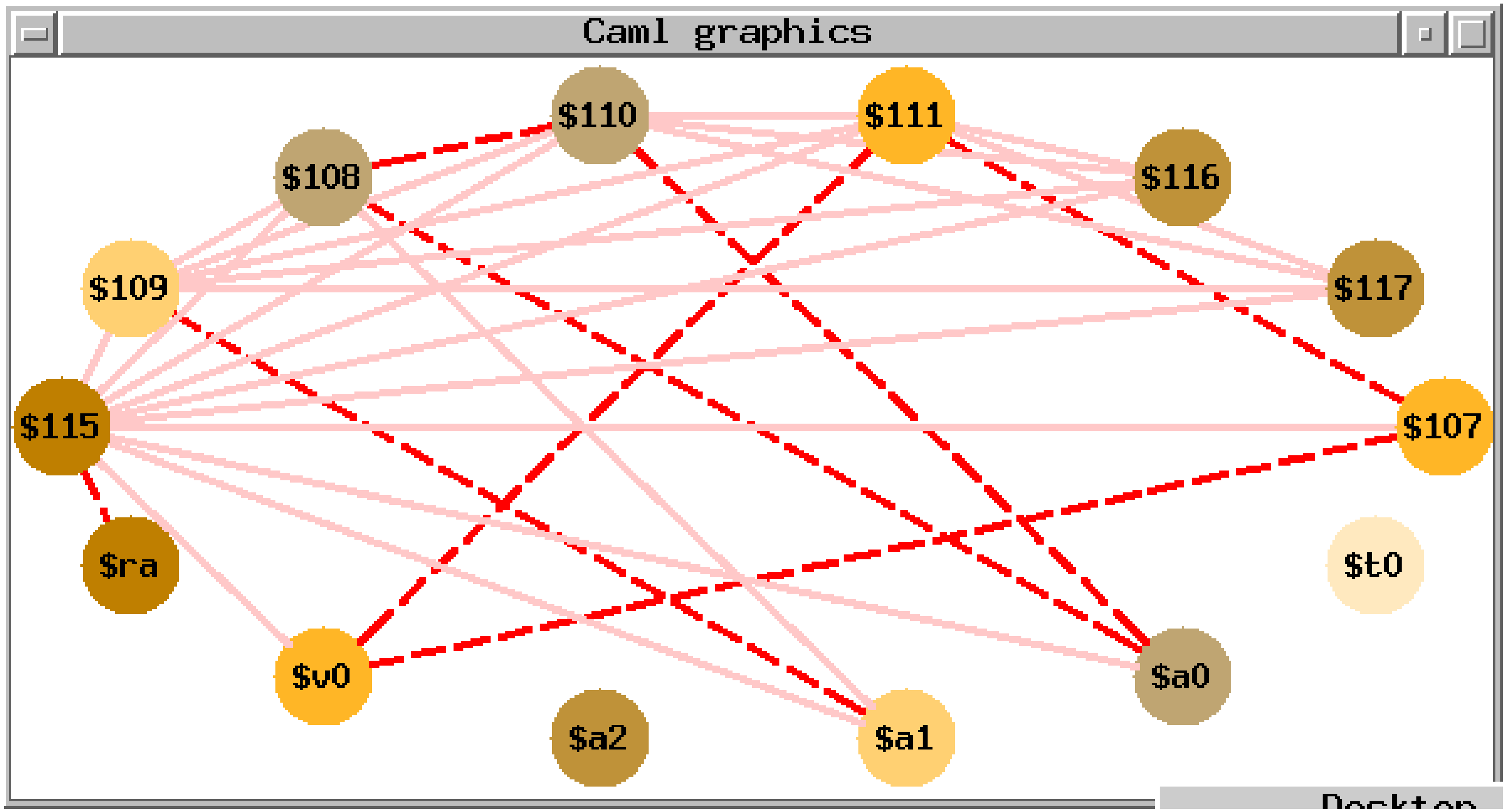
Sélection des instructions

```
power:
      subu $sp, $sp, power_f
      move $115, $ra
      move $116, $s0
      move $117, $s1
       ...
      move $123, $s7
      move $108, $a0
      move $109, $a1
      move $110, $108
      li    $111, 1
      b     L13
L12:
      sra  $124, $109, 1
      sll  $125, $124, 1
      beq $109, $125, L17
L15:
      mul $111, $111, $110
L17:
      sra  $109, $109, 1
      mul $110, $110, $110
L13:
      bgt  $109, $zero, L12
L14:
      move $107, $111
power_end:
      move $v0, $107
      move $ra, $115
      move $s0, $116
      move $s1, $117
       ...
      move $s7, $123
      addu $sp, $sp, power_f
      j     $ra
```

Graphe d'interférence



Allocation de registres



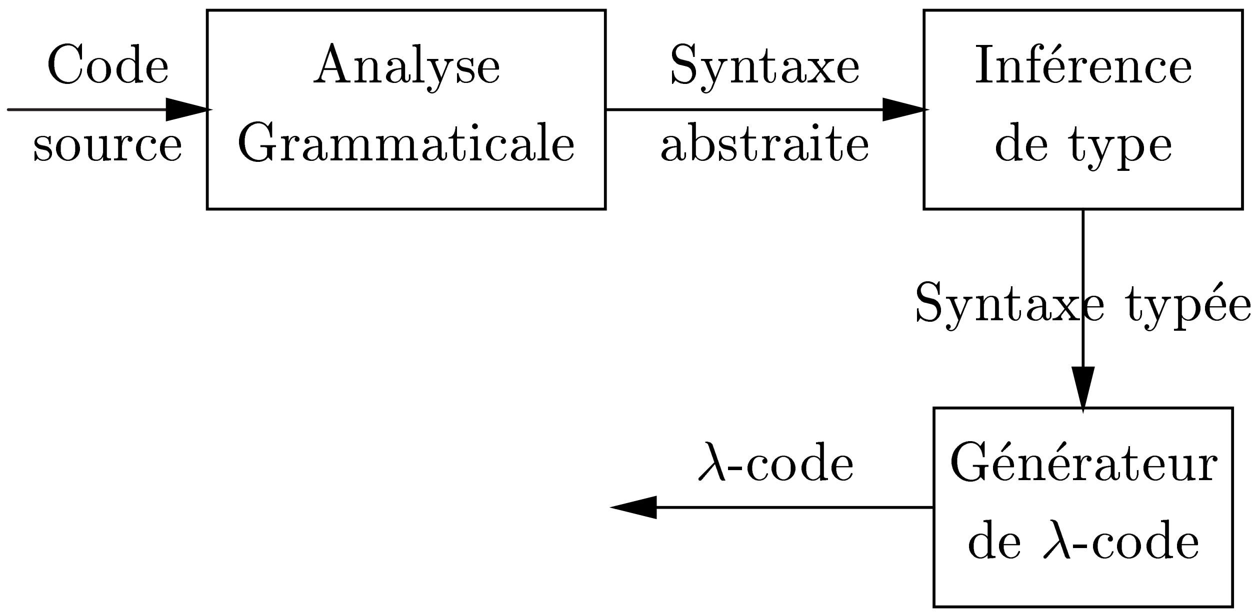
Code final

```
#FUNSTART power
    power_f=0
power:
    li   $v0, 1
    b L13
L12:
    sra $a2, $a1, 1
    sll  $a2, $a2, 1
    beq $a1, $a2, L17
L15:
    mul $v0,$v0,$a0
L17:
    sra $a1, $a1, 1
    mul $a0,$a0,$a0
L13:
    bgt $a1, $zero, L12
L14:
power_end:
    j    $ra
```

La même chose pour Caml

- Il y a beaucoup plus de phases.
- Le modèle des fonctions est différent (langage fonctionnel).
- Le système de modules (ignoré).
- Il y a deux compilateurs (`ocamlc` et `ocamlopt`).
- Compilateur « *bootstrapé* ».
  - Bon test d'expressivité.
  - Facilite le portage.
  - Impact sur la distribution.

Front-end commun



Analyse grammaticale

Techniques traditionelles.

- Lexeur `ocamllex` (rien à signaler).
- Parseur `ocamlyacc` (1444 lignes, gros, 57 shift/reduce).

```
 139  seq_expr : expr
 140           | expr SEMI
 141           | expr SEMI seq_expr
```

```
211: shift/reduce conflict (shift 83, reduce 140) on LET
state 211
seq_expr : expr SEMI .  (140)
seq_expr : expr SEMI . seq_expr  (141)
```

Arbre de syntaxe abstraite enrichi

```
type expression = { pexp_desc: expression_desc; pexp_loc: Location.t }

and expression_desc =
  | Pexp_constant of constant
  ...

%{
let mkexp d =
  { pexp_desc = d; pexp_loc = symbol_rloc() }
%}
simple_expr:
   constant { mkexp(Pexp_constant $1) }
```

Le typeur

Fonctionalité :

- Infère les types.
- Résout les identificateurs ($\rightarrow$ noms uniques).

Arbre de syntaxe typé :

```
type expression =
  { exp_desc: expression_desc; exp_loc: Location.t;
    exp_type: type_expr; exp_env: Env.t }

and expression_desc =
    Texp_ident of Path.t * value_description
  | Texp_constant of constant
  | Texp_let of rec_flag * (pattern * expression) list * expression
```

Un (petit) bout du typeur

Le typage de la séquence « e_1 ; e_2 ».

```
let rec type_exp env sexp = match sexp.pexp_desc with
...
  | Pexp_sequence(sexp1, sexp2) ->
      let exp1 = type_statement env sexp1 in
      let exp2 = type_exp env sexp2 in
      { exp_desc = Texp_sequence(exp1, exp2);
        exp_loc = sexp.pexp_loc;
        exp_type = exp2.exp_type;
        exp_env = env }
```

Suite du petit bout

```
and type_statement env sexp =
  let exp = type_exp env sexp in
  match (expand_head env exp.exp_type).desc with
  | Tarrow _ ->
      Location.prerr_warning
        sexp.pexp_loc Warnings.Partial_application;
      exp
  | Tconstr (p, _, _) when Path.same p Predef.path_unit -> exp
  | Tvar -> exp
  | _ ->
      Location.prerr_warning
        sexp.pexp_loc Warnings.Statement_type;
      exp
```

Le λ-code

C'est le premier « code intermédaire de caml ».

- Fonctions implicites (λ-calcul).
- Structure de bloc (`let`), variables uniques.
- Pattern-matching compilé.

```
type lambda =
    Lvar of Ident.t
  | Lconst of structured_constant
  | Lapply of lambda * lambda list
  | Lfunction of function_kind * Ident.t list * lambda
...
```

Exemple

```
(letrec
  (power/49
    (function n/50 r/51 t/52
     (if (!= n/50 0)
      (if (!= (mod n/50 2) 0)
       (apply power/49 (/ n/50 2) (* r/51 t/52) (* t/52 t/52))
       (apply power/49 (/ n/50 2) r/51 (* t/52 t/52)))
      r/51)))
```

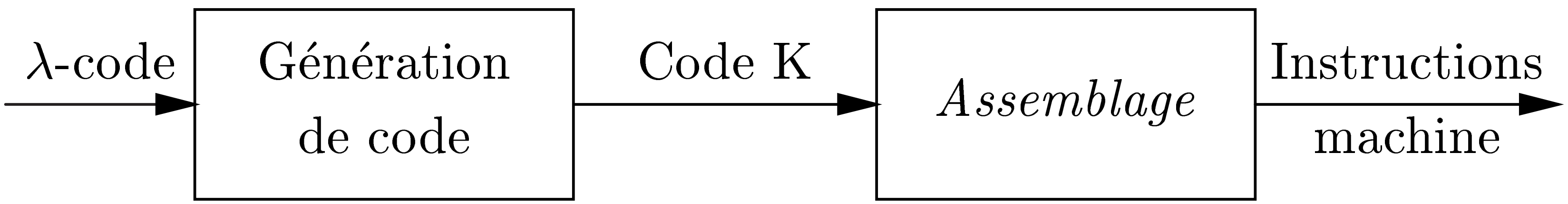
Autre exemple

```
let f op x y =
  let g op = match op with
  | '+' -> x+y
  | '-' -> x-y
  | '*' -> x*y
  | '/' -> x/y
  | _   -> 0 in
  g op
```

λ-code

```
(let (f/49 (function op/50 x/51 y/52
   (let (g/53 (function op/54
      (catch
      (let (switcher/55 (-42+ op/54))
        (if (isout 5 switcher/55) (exit 1)
        (switch* switcher/55
         case int 0: (* x/51 y/52)
         case int 1: (+ x/51 y/52)
         case int 2: (exit 1)
         case int 3: (- x/51 y/52)
         case int 4: (exit 1)
         case int 5: (/ x/51 y/52))))
       with (1) 0)))
     (apply g/53 op/50))))
```

Suite de `ocamlc`



On peut voir le code K comme de l'assembleur (étiquettes), et le code B comme du code machine (suite de mots).

Il y a quelques optimisations entre les deux.

- Les instructions K qui surviennent souvent ensemble sont groupées en une seule instructions machine (`Acc` n `;` `Push` $\rightarrow$ `PUSHACC` n).
- Des instructions compactes traitent quelques cas fréquents d'instructions à argument (`Const` $0 \rightarrow$ `CONST0`)

Extrait de `bytegen.ml`

```
let rec add_pop n cont =
  if n = 0 then cont else
  match cont with
    Kpop m :: cont -> add_pop (n + m) cont
  | _ -> Kpop n :: cont

let rec comp_expr env exp sz cont =
...
  | Llet(str , id , arg, body) ->
       comp_expr env arg sz
        ( Kpush :: comp_expr (add_var id (sz+1) env) body (sz+1)
          ( add_pop 1 cont))
...
```

Example de code K

```
L1: grab 2
    acc 0
    push
    const 0
    neqint
    branchifnot L8
    const 2
    push
    acc 1
    modint
    push
    const 0
    neqint
    branchifnot L9
    acc 2
    push
    acc 3
    mulint
    push
    acc 3
    push
    acc 3
    mulint
    push
    const 2
    push
    acc 3
    divint
    push
    offsetclosure 0
    appterm 3, 6
L9:
    ...
L8  acc 1
    return 3
```

Emission du code

Exemple d'une optimisation.

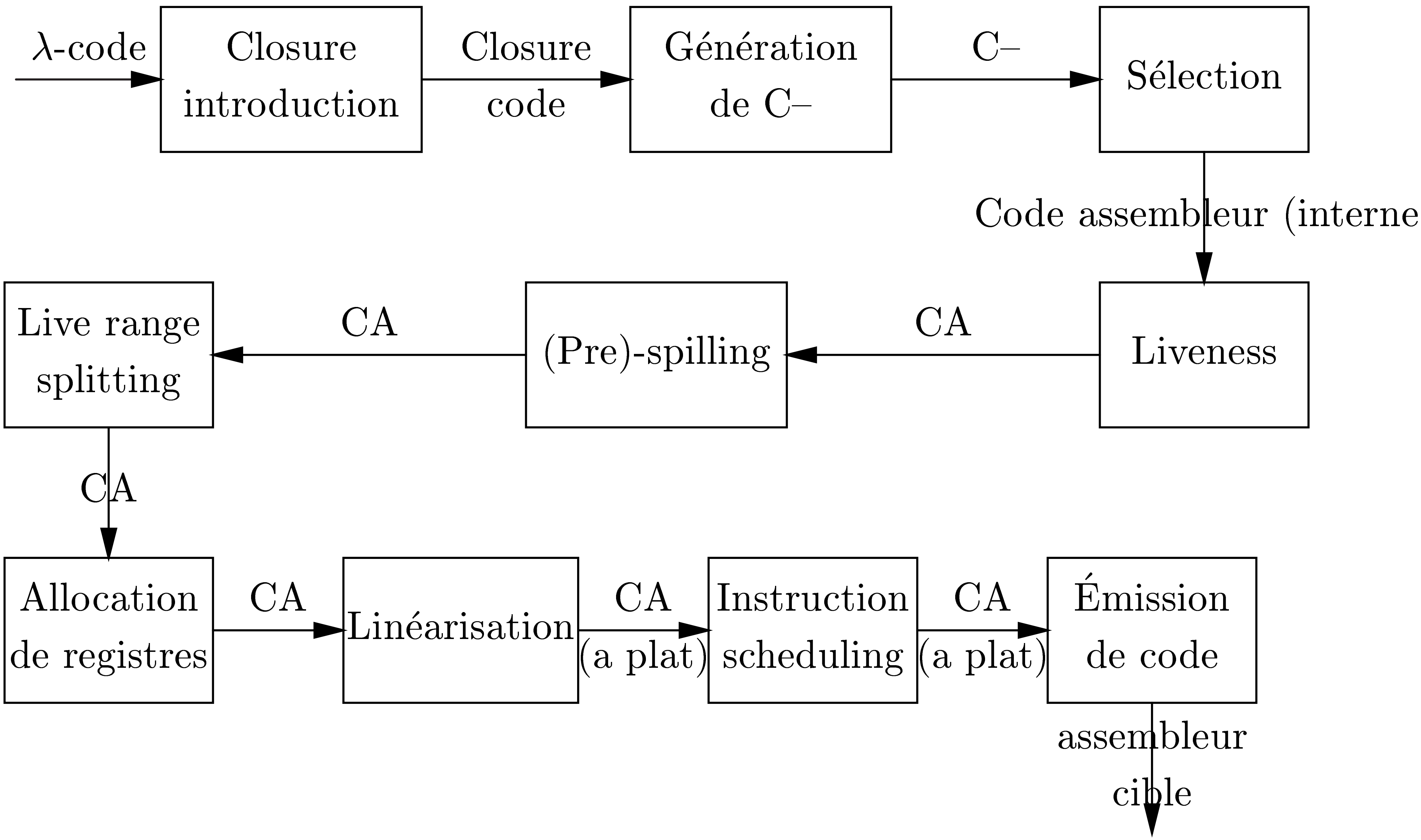
```
let rec emit = function
    []  -> ()
  ...
(* General push then access scheme *)
  | Kpush :: Kacc n :: c ->
      if n < 8 then out(opPUSHACC0 + n) else (out opPUSHACC; out_int n);
      emit c
  ...
(* Default case *)
  | instr  ::  c ->
      emit_instr  instr ;  emit c
```

Autre tâche importante, résolution des étiquettes.

Exemple de code B (désassemblage)

```
7941  GRAB 2
7943  ACC0
7944  BEQ 0, 7975
7947  CONST2
7948  PUSHACC1
7949  MODINT
7950  BEQ 0, 7965
7953  ACC2
7954  PUSHACC3
7955  MULINT
7956  PUSHACC3
7957  PUSHACC3
7958  MULINT
7959  PUSHCONST2
7960  PUSHACC3
7961  DIVINT
7962  PUSHENV0
7963  APPTERM3 6
7965  ACC2
7966  PUSHACC3
7967  MULINT
7968  PUSHACC2
7969  PUSHCONST2
7970  PUSHACC3
7971  DIVINT
7972  PUSHENV0
7973  APPTERM3 6
7975  ACC1
7976  RETURN 3
```

Suite de `ocamlopt`



Quelques détails

Closure introduction Les fermetures sont explicitées, ainsi que les appels directs. Optimisations d'*inlining* et de *constant folding* (interprétation à la compilation).

C- - Est le code intermédiaire que nous connaissons. Il doit en plus expliciter l'interaction avec le GC.

Assembleur interne

- Machine idéale, mais le code est encore structuré.
- Des instructions spécifiques sont ajoutées, selon la cible.
- Le selecteur est générique, il peut être précisé selon la machine ciblée (programmation objet).

Allocation de registre L'algorithme est le même, la réalisation est différente.

Exemple de power (récursif)

```
let rec power x n = match n with
| 0 -> 1
| _ -> let t = power x (n/2) in
       let t = t*t in
       if n mod 2 = 0 then t
       else x * t
```

```
(letrec
  (power/49
   (function x/50 n/51
   (if (!= n/51 0)
     (let (t/52 (apply power/49 x/50 (/ n/51 2)) t/53 (* t/52 t/52))
     (if (== (mod n/51 2) 0) t/53 (* x/50 t/53)))
     1)))
```

C--

```
(function Power_power_49 (x/50: addr n/51: addr)
 (if (!= n/51 1)
   (let
     (t/52 (app "Power_power_49" x/50 (+ (<< (/ (>>s n/51 1) 2) 1) 1) addr)
      t/53 (+ (* (+ t/52 -1) (>>s t/52 1)) 1))
     (if (== (+ (<< (mod (>>s n/51 1) 2) 1) 1) 1) t/53
       (+ (* (+ x/50 -1) (>>s t/53 1)) 1)))
   3))
```

Noter l'introduction des opérations « $2n+1$ ». La caractérisation `addr`.

Code assembleur interne

```
Power_power_49(R/0[%eax]
  R/1[%ebx])
  x/8 := R/0[%eax]
  n/9 := R/1[%ebx]
  if n/9 !=s 1 then
    I/11 := n/9
    I/11 := I/11 >>s 1
    I/12 := I/11
    I/12 := I/12 div 2
    I/13 := I/12  * 2 + 1
    R/0[%eax] := x/8
    R/1[%ebx] := I/13
    R/0[%eax] :=
     call "Power_power_49"
       R/0[%eax]
       R/1[%ebx]
    t/14 := R/0[%eax]
    I/15 := t/14
    I/15 := I/15 >>s 1
    I/16 := t/14
    I/16 := I/16 + -1
    I/17 := I/16
    I/17 := I/17 * I/15
    t/18 := I/17
    t/18 := t/18 + 1
    I/19 := n/9
    I/19 := I/19 >>s 1
    R/3[%edx] := I/19
    R/3[%edx] := R/3[%edx] mod 2
    I/20 := R/3[%edx]
    I/21 := I/20  * 2 + 1
    if I/21 ==s 1 then
      R/0[%eax] := t/18
      return R/0[%eax]
    else
      I/22 := t/18
      I/22 := I/22 >>s 1
      I/23 := x/8
      I/23 := I/23 + -1
      I/24 := I/23
      I/24 := I/24 * I/22
      I/25 := I/24
      I/25 := I/25 + 1
      R/0[%eax] := I/25
      return R/0[%eax]
    endif
  else
    I/10 := 3
    R/0[%eax] := I/10
    return R/0[%eax]
  endif
```

Noter, les conventions d'appel, les idiomes (deux adresses, `I * 2 + 1`).

Après spill préalable

```
Power_power_49(R/0[%eax]
  R/1[%ebx])
  x/8 := R/0[%eax]
  n/9 := R/1[%ebx]
  if n/9 !=s 1 then
    spilled-x/26 := x/8 (spill)
    spilled-n/27 := n/9 (spill)
    I/11 := n/9
    I/11 := I/11 >>s 1
    I/12 := I/11
    I/12 := I/12 div 2
    I/13 := I/12  * 2 + 1
    R/0[%eax] := x/8
    R/1[%ebx] := I/13
    R/0[%eax] :=
     call
       "Power_power_49"
         R/0[%eax]
         R/1[%ebx]
    t/14 := R/0[%eax]
    I/15 := t/14
    I/15 := I/15 >>s 1
    I/16 := t/14
    I/16 := I/16 + -1
    I/17 := I/16
    I/17 := I/17 * I/15
    t/18 := I/17
    t/18 := t/18 + 1
    n/28 := spilled-n/27 (reload)
    I/19 := n/28
    I/19 := I/19 >>s 1
    R/3[%edx] := I/19
    R/3[%edx] := R/3[%edx] mod 2
    I/20 := R/3[%edx]
    I/21 := I/20  * 2 + 1
    if I/21 ==s 1 then
      R/0[%eax] := t/18
      return R/0[%eax]
    else
      I/22 := t/18
      I/22 := I/22 >>s 1
      x/29 := spilled-x/26 (reload)
      I/23 := x/29
      I/23 := I/23 + -1
      I/24 := I/23
      I/24 := I/24 * I/22
      I/25 := I/24
      I/25 := I/25 + 1
      R/0[%eax] := I/25
      return R/0[%eax]
    endif
  else
    I/10 := 3
    R/0[%eax] := I/10
    return R/0[%eax]
  endif
```

Après allocation de registres

```
Power_power_49(R/0[%eax]
  R/1[%ebx])
  x/8[%eax] := R/0[%eax]
  n/9[%ebx] := R/1[%ebx]
  if n/9[%ebx] !=s 1 then
    spilled-x/26[s1] := x/8[%eax] (spill)
    spilled-n/27[s0] := n/9[%ebx] (spill)
    I/11[%ebx] := n/9[%ebx]
    I/11[%ebx] := I/11[%ebx] >>s 1
    I/12[%ebx] := I/11[%ebx]
    I/12[%ebx] := I/12[%ebx] div 2
    I/13[%ebx] := I/12[%ebx]  * 2 + 1
    R/0[%eax] := x/8[%eax]
    R/1[%ebx] := I/13[%ebx]
    R/0[%eax] :=
      call "Power_power_49"
        R/0[%eax]
        R/1[%ebx]
    t/14[%eax] := R/0[%eax]
    I/15[%ebx] := t/14[%eax]
```

```
I/15[%ebx] := I/15[%ebx] >>s 1
I/16[%eax] := t/14[%eax]
I/16[%eax] := I/16[%eax] + -1
I/17[%eax] := I/16[%eax]
I/17[%eax] := I/17[%eax] * I/15[%ebx]
t/18[%ebx] := I/17[%eax]
t/18[%ebx] := t/18[%ebx] + 1
n/28[%edx] := spilled-n/27[s0] (reload)
I/19[%edx] := n/28[%edx]
I/19[%edx] := I/19[%edx] >>s 1
R/3[%edx] := I/19[%edx]
R/3[%edx] := R/3[%edx] mod 2
I/20[%edx] := R/3[%edx]
I/21[%ecx] := I/20[%edx]  * 2 + 1
if I/21[%ecx] ==s 1 then
  R/0[%eax] := t/18[%ebx]
  return R/0[%eax]
else
 ...
```

Code final

```
Power_power_49:
        subl    $8, %esp
.L102:
        cmpl    $1, %ebx
        je      .L100
        movl    %eax, 4(%esp)
        movl    %ebx, 0(%esp)
        sarl    $1, %ebx
        testl   %ebx, %ebx
        jge     .L103
        addl    $1, %ebx
.L103:  sarl    $1, %ebx
        lea     1(, %ebx, 2), %ebx
        call    Power_power_49
.L104:
        movl    %eax, %ebx
        sarl    $1, %ebx
        decl    %eax
        imull   %ebx, %eax
        movl    %eax, %ebx
        incl    %ebx
        movl    0(%esp), %edx
        sarl    $1, %edx
        movl    %edx, %eax
        testl   %eax, %eax
        jge     .L105
        addl    $1, %eax
.L105:  andl    $-2, %eax
        subl    %eax, %edx
        lea     1(, %edx, 2), %ecx
        cmpl    $1, %ecx
        jne     .L101
```

```
        addl    $8, %esp
        movl    %ebx, %eax

ret
        .align  16
.L101:
        sarl    $1, %ebx
        movl    4(%esp), %eax

        decl    %eax
        imull   %ebx, %eax
        incl    %eax
        addl    $8, %esp
        ret
        .align  16
.L100:
        movl    $3, %eax
        addl    $8, %esp
        ret
```

Noter les rattrapages de dernière minute.

Code C équivalent

```
power:
        pushl %ebp
        movl %esp,%ebp
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%ebx
        movl 12(%ebp),%esi
        testl %esi,%esi
        jne .L17
        movl $1,%eax
        jmp .L21
        .p2align 4,,7
.L17:
        movl %esi,%eax
        shrl $31,%eax
        addl %esi,%eax
        sarl $1,%eax
        pushl %eax
        pushl %ebx
        call power
        testl $1,%esi
        je .L19
        imull %eax,%ebx
        imull %eax,%ebx
        movl %ebx,%eax
        jmp .L21
        .p2align 4,,7
.L19:
```

.L21:

imull %eax,%eax

leal

−8(%

ebp

),%

esp

popl %ebx

popl %esi

leave

ret