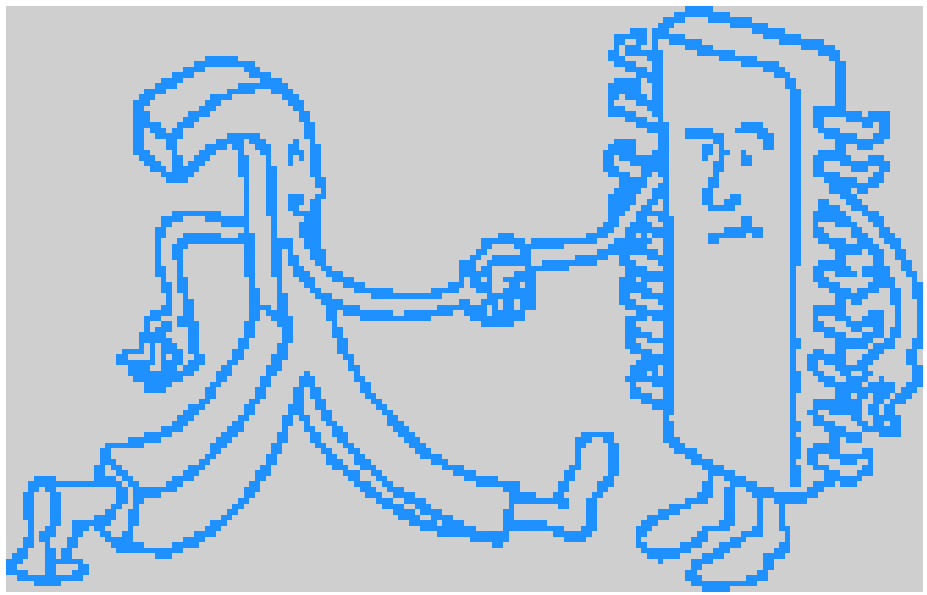
COMPILATION (3I018) : COURS 6

Machine virtuelle MV_{3I018} : conception et implantation



Emmanuel Chailloux

Plan du cours 6

- Caratéristiques générales
- Instructions
- Structure générique pour les valeurs allouées & environnements
- Appels de primitives
- Appel de fonctions
  - représentation des valeurs fonctionnelles
  - cadre d'appel d'une fonction
  - instructions :PUSH FUN ..., CALL et RETURN
  - exemples
- Enrichir la bibliothèque d'exécution
- Assembleur et byte-code

Présentation de la machine MV_{3I018}

MV_{3I018} est une machine à pile

- représentation uniforme des valeurs manipulées
- une machinerie : du code et des zones mémoire
- peu d'instructions : une quinzaine mais des primitives pour les calculs d'expression
- manipulant des valeurs fonctionnelles
- et ayant un récupérateur automatique de mémoire

Représentation des valeurs MV_{3I018} (1)

nécessité de conserver une information de type sur les valeurs :

- des valeurs immédiates (représentées par des entiers)
  - des entiers, des booléens (`TRUE` et `FALSE`), des numéros de primitives
- des valeurs allouées dans le tas
  - des blocs de taille connue
  - des valeurs fonctionnelles

Machinerie MV_{3I018} (1) : zones mémoire

un compteur ordinal *pc* (index de l'instruction à exécuter)
et un pointeur de pile *sp* et des instructions manipulant :

- du code
- des primitives
- un environnement global
- une pile
- un tas
- des environnements locaux (cadres d'appel pour les fonctions et primitives)

Les globales et la pile sont vues comme un vecteur (`varray_t`).

Instructions

Instructions (1) : Environnement global

- GALLOC : allocation d'une variable globale
- GFETCH n : lecture de la variable globale numéro n
- GSTORE n : affectation de la variable globalenuméro n

Instructions (2) : Environnement local

Environnement local : mêmes fonctionnalités:

- FETCH n : lecture de la variable locale numéro n
- STORE n : affectation de la variable locale numéro n

voir prochain cours

Instructions (3) : Opération sur la pile

- POP : dépilement

```
1   // de'piler
2   case I_POP: {
3     value_t * val = varray_pop(vm->stack);
4     break; }
```

- PUSH *type val* : empilement de la valeur *val* de type *type* ;

```
1     // empilement d'une valeur
2   case I_PUSH: {
3     value_t value;
4     switch(vm_next(vm)) {
5       case T_INT: // placer un entier
6         value_fill_int(&value, vm_next(vm));
7         break;
8         case T_PRIM: // placer un nume'ro de primitive
9         value_fill_prim(&value, vm_next(vm));
10        break;
11        // ...
12      }
13    varray_push(vm->stack,&value);
14    break;
15  }
```

Instructions (4) : cadre d'appel

cadre d'appel: :

- CALL *a* : appel d'un fonction ou d'une primivite d'arité *a*
- RETURN : retour de fonction : destruction du cadre d'appel courant

création d'un nouveau cadre d'appel pour le CALL et destruction de cadre courant au RETURN

voir prochain cours

Instructions (5) : opération de contrôle

opérations de contrôle: : modification du compteur ordinal *pc*

- saut inconditionnel

```
1     // saut inconditionnel
2   case I_JUMP:
3     vm->frame->pc = vm->program->bytecode[vm->frame->pc];
4     break;
```

- saut conditionnel

```
1    // si le sommet de pile est faux, alors on effectue le saut,
2     // sinon on de'pile simplement
3   case I_JFALSE:
4     if(value_is_false(varray_pop(vm->stack))) {
5       vm->frame->pc = vm->program->bytecode[vm->frame->pc];
6     } else {
7       vm_next(vm);
8     }
9     break;
```

Instructions (6) : primitives (1)

- un mécasnime général

```
1  void execute_prim(vm_t * vm, varray_t *stack, int prim, int n) {
2    switch(prim) {
3      // Les fonctions arithme'tiques sont traite'es d'un coup.
4    case P_ADD:
5    case P_SUB:
6    case P_MUL:
7    case P_DIV:
8      do_arith_prim(stack, n, prim); break;
9  // ...
```

- factorisé pour les primitives arithmétiques

```
1  void do_arith_prim(varray_t *stack, int n, int op) {
2   int r,i ;
3    r = value_int_get(varray_top(stack));
4    for(i=1; i<n; i++) {
5      r = apply_arith_prim(op, r,value_int_get(varray_top_at(stack, i)));
6    }
7    varray_popn(stack, n-1);
8   }
9   value_fill_int(varray_top(stack), r);
10 }
```

Instructions (6) : primitives (2)

- application effective :

```
1  static int apply_arith_prim(int prim, int n1, int n2) {
2    switch(prim) {
3    case P_ADD: return (n1 + n2);
4    case P_SUB: return (n1 - n2);
5    case P_MUL: return (n1 * n2);
6   // ...
```

- numérotation dans le fichier `constants.h` créé par le compilateur :

```
1  /* Constantes pour les primitives */
2  /** primitive + */
3  #define P_ADD 0
4  /** primitive - */
5  #define P_SUB 1
6  /** primitive * */
7  #define P_MUL 2
```

Exemples (1)

- programme addition de deux variables :

```
1  var a = 10 ;
2  var b = 20 ;
3  a + b ;
```

- byte-code produit

```
1    GALLOC
2    PUSH INT 10
3    GSTORE 0
4    GALLOC
5    PUSH INT 20
6    GSTORE 1
7    GFETCH 1
8    GFETCH 0
9    PUSH PRIM 0
10   CALL 2
11   POP
```

Exemples (2)

- conditionnelle :

```
1  if (true) {
2      42;
3  } else {
4      38;
5  }
```

- byte-code produit

```
1    PUSH BOOL TRUE
2    JFALSE L1
3    PUSH INT 42
4    POP
5    JUMP L2
6  L1:
7    PUSH INT 38
8    POP
9  L2:
```

Rappel : valeurs de la MV_{3I018}

déclaration des types:

```
1  /** Donne'es associe'es a' une valeur */
2
3  union _value_data {
4    int  as_int;   /*!< si entier (T_INT), No de primitive (T_PRIM),
5                        ou boole'en (T_BOOL) */
6    struct _block *as_block;    /*!< si c'est un bloc (T_BLOCK) */
7    closure_t  as_closure; /*!< si c'est une fermeture (T_CLOSURE) */
8  };
9
10 /** Repre'sentation d'une valeur. */
11
12 typedef struct {
13   int            type;  /*!< le type de la valeur */
14   union _value_data data; /*!< les donne'es supple'mentaires,
15                              en fonction du type. */
16 } value_t;
```

Rappel : valeurs de la MV_{3I018} (1)

allocation: Une valeur est une `value_t` allouée dans la pile C :

```
1  case I_PUSH: {
2      value_t value;
3  \\
4      switch(vm_next(vm)) {
5      case T_INT: // placer un entier
6        value_fill_int(&value, vm_next(vm));
7        break;
8  \\ ...
```

```
1  /** Pre'paration d'une valeur de type entier.
2   * \param[in,out] value la valeur a' pre'parer
3   * \param value la valeur entie're de la valeur.
4   */
5  void value_fill_int(value_t *value,  int num) {
6    value->type = T_INT;
7    value->data.as_int = num;
8  }
```

Rappel : valeurs de la MV_{3I018} (2)

allocation: Seules les fermetures (et d'autres structures allouées) devront être allouées dans le tas de la machine virtuelle

```
1  case I_PUSH: {
2      value_t value;
3  \\
4      case T_FUN: { // placer une fermeture
5          closure_t closure;
6          closure.env = vm->frame->env; // on capture l'environnement courant
7          closure.pc = vm_next(vm); // le PC de la fermeture est la prochaine ↩
               information
8          value_fill_closure(&value, closure);
9        }
10       break;
11 \\ ...
```

```
1  void value_fill_closure(value_t *value, closure_t closure) {
2    value->type = T_FUN;
3    value->data.as_closure = closure;
4  }
```

Structure générique pour valeurs allouées (1)

basée sur les varray (varray.h)

```
1  /** Structure ge'ne'rique pour tout tableau de cellules.
2   */
3  typedef struct {
4    value_t* content; /*!< le contenu (tableau dynamique de cellules) */
5    unsigned int capacity; /*!< capacite' du tableau (taille alloue'e) */
6    unsigned int top; /*!< dernier e'le'ment utilise' (ou sommet de pile).
7                       * Remarque : tous les e'le'ments entre top+1 et capacity
                            -1 sont inutilise's (mais alloue's) */
8  } varray_t;
```

```
1  varray_t *varray_allocate(unsigned int initial_capacity) {
2    varray_t *res = (varray_t *) malloc(sizeof(varray_t));
3    assert(initial_capacity >= 0);
4    res->content  = (value_t *) malloc(sizeof(value_t) * initial_capacity);
5    assert(res->content!=NULL);
6    res->capacity = initial_capacity;
7    res->top      = 0; // le marqueur top a' 0 indique qu'il n'y a aucun e'le'
         ent (taille 0)
8    return res;
9  }
```

Structure générique pour valeurs allouées (2)

principales fonctions :

```
1  void varray_expandn(varray_t *varray, unsigned int n);
2  void varray_popn(varray_t *varray, unsigned int n);
3  value_t *varray_at(varray_t *varray, unsigned int n);
4  void varray_set_at(varray_t *varray, unsigned int n, value_t *value);
5  value_t *varray_top_at(varray_t *varray, unsigned int n);
6  value_t *varray_top(varray_t *varray);
7  void varray_set_top(varray_t *varray, value_t *value);
8  void varray_set_top_at(varray_t *varray, unsigned int n, value_t *value);
9  void varray_push(varray_t *varray, value_t *value);
10 value_t *varray_pop(varray_t *varray);
11 int varray_size(varray_t *varray);
12 int varray_empty(varray_t *varray);
13 void varray_destroy(varray_t *varray);
14 void varray_print(varray_t *varray);
15 void varray_stack_print(varray_t *varray);
```

```
1  void varray_set_at(varray_t *varray, unsigned int n, value_t *value) {
2    // pre'condition : l'index doit exister
3    assert(n<varray->top);
4    *(varray_at(varray,n)) = *value;
5  }
```

Rappel : représentation de la machine

- la struct vm_t :

```
1  typedef struct _vm {
2    int debug_vm; /*!< VM en mode debug (1) ou non (0) */
3    varray_t *globs; /*!< l'environnement global (variables globales) */
4    varray_t *stack;  /*!< la pile */
5    frame_t *frame; /*!< la fene^tre d'entre'e */
6    program_t *program;
7    gc_t *gc;
8  } vm_t;
```

- l'initialisation (version simplifiée)

```
1  vm_t * init_vm(program_t *program) {
2    vm_t * vm  = (vm_t *) malloc(sizeof(vm_t));
3    vm->program = program;
4    vm->globs = varray_allocate(GLOBS_SIZE);
5    varray_expandn(vm->globs,1);
6    vm->stack = varray_allocate(STACK_SIZE);
7    vm->frame = frame_push(NULL, // pas de call frame parente
8                           NULL, // environnement local vide
9                           0,  // de'but de pile ... au de'but
10                          0); // commencer par la premie're instruction
11   vm->gc = init_gc();
12   return vm;}
```

Environnement global

- un tableau global
- opérations : GSTORE et GFETCH

```
1     // de'piler le sommet de pile et le placer au bon endroit dans l'↩
          environnement global
2   case I_GSTORE:
3     varray_set_at(vm->globs,
4                   vm_next(vm),
5                   varray_pop(vm->stack));
6     break;
7
8     // empiler la valeur d'une variable globale
9   case I_GFETCH:
10    varray_push(vm->stack,
11                varray_at(vm->globs, vm_next(vm)));
12    break;
```

Environnement local

- liste chaînée de tableau pour les déclarations locales :

```
1  /** Structure d'un environnement (variables locales).
2   */
3  typedef struct _env {
4    int gc_mark;  /*!< marque pour le GC */
5    varray_t * content; /*!< contenu de l'environnement (tableau de ↩
         cellules). */
6    struct _env *next; /*!< environnement suivant (ou "englobant")  */
7  } env_t;
```

- opérations FETCH et STORE :

```
1   // de'piler le sommet de pile et le sauvegarder dans l'environnement ↩
        local
2    case I_STORE:
3      env_store(vm->frame->env, vm_next(vm),
4                      varray_pop(vm->stack));
5      break;
6  // empiler la valeur d'une variable locale
7    case I_FETCH:
8      varray_push(vm->stack,  // et on recopie
9                  env_fetch(vm->frame->env, vm_next(vm)));
10     break;
```

Appel de valeurs fonctionnelles

- primitives
- fonction sans variables libres
  - à un paramètre
  - à plusieurs paramètres
  - récursive
- cas général : valeur fonctionnelle
  - valeur fonctionnelle comme paramètre
  - déclaration locale d'une fonction, extension de portée d'une variable locale
  - retour d'une valeur fonctionnelle

Appel d'une primitive (1)

- les arguments sont sur la pile (`PUSH`)
- la primitive est au sommet de pile (`PUSH PRIM 0`)
- on appelle avec la bonne arité (`CALL 2`)

```
1  1 + 4;
```

```
1  PUSH INT 4
2  PUSH INT 1
3  PUSH PRIM 0
4  CALL 2
5  POP
```

ici on empile 4 (`PUSH 4`)
puis 1 (`PUSH INT 1`)
puis + (`PUSH PRIM 0`)
et enfin l'appel d'arité 2 est effectué (`CALL 2`)

Appel d'une primitive (2)

```
1   case I_CALL: {
2     // re'cupe'rer la fermeture ou la primitive
3     value_t *fun = varray_pop(vm->stack);
4     int nb_args = vm_next(vm);
5
6     switch(fun->type) {
7   //...
8     case T_PRIM: {
9       // nume'ro de primitive encode'e dans la valeur.
10      int prim_num = value_prim_get(fun);
11      // exe'cuter la primitive (aie)
12      execute_prim(vm, vm->stack, prim_num, nb_args);
13      break;
14    }
15  // ...
16  } }
```

Appel d'une primitive (3) - version simplifiée

```
1  void execute_prim(vm_t * vm, varray_t *stack, int prim, int n) {
2    switch(prim) {
3       // Les fonctions arithme'tiques sont traite'es d'un coup.
4    case P_ADD:   case P_SUB:   case P_MUL:   case P_DIV:
5      do_arith_prim(stack, prim, 2); break;
6   \\ ...
```

```
1  void do_arith_prim(varray_t *stack, int prim, int n) {
2    value_t value;  value_t  *value1;  value_t  *value2;  int r;
3    switch(n) {
4    case 2 :
5      value1 = varray_pop(stack);    value2 = varray_pop(stack);
6      r = apply_arith_prim(prim,value_int_get(value1),value_int_get(value2));
7      value_fill_int(&value,r);
8      varray_push(stack,&value);
9      break;
10   \\ ...
```

```
1  static int apply_arith_prim(int prim, int n1, int n2) {
2    switch(prim) {
3      case P_ADD: return (n1 + n2);
4  \\ ...
```

Représentation des valeurs fonctionnelles

- Comme toute valeur de la MV_{3I018} avec un type et une donnée spécifiques :

```
1  /** Donne'es associe'es a' une valeur */
2  union _value_data {
3    int             as_int;     /*!< si entier (T_INT), No de primitive (↩
         T_PRIM), ou boole'en (T_BOOL) */
4    closure_t  as_closure; /*!< si c'est une fermeture (T_FUN) */
5  };
```

- contenant l'environnement lexical nécessaire au calcul du corps de la fonction :

```
1  /** Structure pour les fermetures.
2   */
3  typedef struct {
4    int         pc;  /*!<  Compteur de programme pour le corps de la ↩
         fermeture. */
5    struct _env *env;  /*!<  Environnement lexical capture' par la ↩
         fermeture. */
6  } closure_t;
```

Cadre d'appel d'une fonction (1)

Représentation des cadres d'appel (call frame ou encore blocs d'activation) de fonction. Un cadre d'appel est associé à chaque appel de fonction (ou de fermeture).

- un environnement local pour les variables lexicales (potentiellement chaîné avec un environnement englobant)
- une zone de pile pour stocker les résultats intermédiaires des calculs.
- le compteur de programme de l'appelant (pour pouvoir retourner juste après l'appel).
- le cadre d'appel de l'appelant (au premier appel : cadre de pile du top-niveau).

```
1  typedef struct _frame {
2    env_t *env;    /*!< l'environnement lexical du cadre d'appel. */
3    unsigned int sp;   /*!< le pointeur de pile */
4    unsigned int pc;    /*!< le PC de l'appelant pour le retour de fonction */
5    struct _frame *caller_frame; /*!< le cadre d'appel de l'appelant (ou cadre ↩
         parent) */
6  } frame_t;
```

Cadre d'appel d'une fonction (2)

structure de pile des cadres d'appel:

```
1  frame_t *frame_push(frame_t *frame,
2                      env_t *env,
3                      unsigned int sp,
4                      unsigned int pc) {
5    frame_t *res = (frame_t *) malloc(sizeof(frame_t));
6    assert(res!=NULL);
7
8    res->sp = sp;
9    res->env = env;
10   res->pc = pc;
11   res->caller_frame = frame;
12
13   return res;
14 }
15
16 frame_t *frame_pop(frame_t *frame) {
17   frame_t *caller_frame = frame->caller_frame;
18
19   free(frame); // puis de'truire le cadre de pile
20
21   return caller_frame;
22 }
```

Appel d'une fonction : `PUSH FUN label`

Empiler une valeur fonctionnelle :

- créér un couple de type `closure_t` : environnement + code
  - l'environnement est celui du cadre d'appel
  - le code correspond au corps de la fonction
- puis en faire une valeur (`value_t`)

```
1   case I_PUSH: {
2     value_t value;
3   //...
4     switch(vm_next(vm)) {
5   //...
6     case T_FUN: { // placer une fermeture
7     closure_t closure;
8         closure.env = vm->frame->env; // on capture l'environnement courant
9         closure.pc = vm_next(vm); // le PC de la fermeture est la prochaine ↩
                instruction
10        value_fill_closure(&value, closure);
11      }
12      break;
13  //...
```

Appel d'une fonction : CALL *i*

- allouer l'environnement de la fermeture
- empiler un nouveau cadre d'appel

```
1  case I_CALL: { // appeler une fermeture (fonction) ou une primitive
2   // re'cupe'rer la fermeture ou la primitive
3   value_t *fun = varray_pop(vm->stack);
4   int nb_args = vm_next(vm);
5   switch(fun->type) {
6       // si c'est une fermeture
7     case T_FUN: {
8       int i;
9       closure_t closure = value_closure_get(fun);
10      env_t *env = gc_alloc_env(vm->gc, nb_args, closure.env);
11      // recopier les arguments de la pile vers l'environnement local
12      // de la fermeture
13      for (i=0; i<nb_args; i++) {
14       varray_set_at(env->content, i,varray_top_at(vm->stack, i));
15      }
16      varray_popn(vm->stack, nb_args); // tout de'piler
17     // empiler une nouvelle call frame.
18      vm->frame = frame_push(vm->frame, env, vm->stack->top, vm->frame->pc);
19      vm->frame->pc = closure.pc;
20      break;  }
```

Appel d'une fonction : RETURN

- récupération du résultat (au sommet de la pile)
- dépilement pour revenir au niveai de l'appel
- empiler le résultat
- revenir au cadre d'appel précédent (avec le bon pc)

```
1    // retour de fonction
2   case I_RETURN: {
3    // la pile contient la valeur de retour au sommet [res ...]
4    value_t *res = varray_pop(vm->stack);
5
6    // il faut se de'placer dans le bon sens
7    assert(vm->stack->top>=vm->frame->sp);
8
9    vm->stack->top = vm->frame->sp;
10   varray_push(vm->stack, res);
11   vm->frame = frame_pop(vm->frame);
12  }
13   break;
```

Appel d'une fonction : prog1.js

```
1  function succ(x) {return (x+1);}
2  succ(5);
```

```
1  var f = lambda(x){return x + 1;};
2  f(5);
```

```
1    GALLOC
2    JUMP L2
3  L1:
4    PUSH INT 1
5    FETCH 0
6    PUSH PRIM 0
7    CALL 2
8    RETURN
9    PUSH UNIT
10   RETURN
11 L2:
12   PUSH FUN L1
13   GSTORE 0
14   PUSH INT 5
15   GFETCH 0
16   CALL 1
17   POP
```

- label `L1` : code de la fonction succ
  - empiler 1 (`PUSH INT 1`), empiler la valeur de x (`FETCH`),
  - empiler $+$ (`PUSH PRIM 0`), appel d'arité 12 (`CALL 2`), (`RETURN`)
- label `L2` : code de l'appel `succ(5)`
  - empiler succ (`PUSH FUN L1`), stocker en global
  - empiler 5 (`PUSH INT 5`), empiler la globale 0 (`GFETCH 0`)
  - appel d'arité 1 (`CALL 1`)
- saut direct vers `L2` au début de programme

Appel d'une fonction à plusieurs paramètres

```
1  function discr(a,b,c) {
2    return (b * b - 4 * c);
3  }
4
5  discr(1,2,1);
```

```
1  var discr = lambda(a,b,c) {
2    return (b * b - 4 * c);
3  }
4
5  discr(1,2,1);
```

```
1    GALLOC
2    JUMP L2
3  L1:
4    FETCH 2
5    PUSH INT 4
6    PUSH PRIM 2
7    CALL 2
8    FETCH 1
9    FETCH 1
10   PUSH PRIM 2
11   CALL 2
12   PUSH PRIM 1
13   CALL 2
14   RETURN
15   PUSH UNIT
16   RETURN
```

```
17 L2:
18   PUSH FUN L1
19   GSTORE 0
20   PUSH INT 1
21   PUSH INT 2
22   PUSH INT 1
23   GFETCH 0
24   CALL 3
25   POP
```

- `discr` dans l'env global : `PUSH FUN L1` puis `GSTORE`
- empilement des arguments `PUSH INT` *i*
- empilement `discr` : `GFETCH 0`
- appel arité 3 : `CALL 3`

Appel d'une fonction récursive

```
1  function power(a,n) {
2   if (n == 0) {return 1;}
3   else {
4    return (a *
5            power(a,n-1));
6   }
7  }
8
9  power(2,8);
```

- L2 : appel de power
  - GSTORE 0
  - GFETCH 0
- L1 : définition
  - L1 : partie if-then
  - L3 : partie else
  - *a* : FETCH 0
  - *n* : FETCH 1
- L4 : code mort

version

```
1   GALLOC
2   JUMP L2
3  L1:
4   PUSH INT 0
5   FETCH 1
6   PUSH PRIM 4
7   CALL 2
8   JFALSE L3
9   PUSH INT 1
10  RETURN
11  JUMP L4
12 L3:
13  PUSH INT 1
14  FETCH 1
15  PUSH PRIM 1
16  CALL 2
17  FETCH 0
```

corrigée

```
18  GFETCH 0
19  CALL 2
20  FETCH 0
21  PUSH PRIM 2
22  CALL 2
23  RETURN
24 L4:
25  PUSH UNIT
26  RETURN
27 L2:
28  PUSH FUN L1
29  GSTORE 0
30  PUSH INT 8
31  PUSH INT 2
32  GFETCH 0
33  CALL 2
34  POP
```

Passage de fonction comme paramètre (1)

```
1  function succ (x) { return (x+1); }
2  function applyN(f,n,x) {
3    if (n == 0) { return x;}
4    else { return(applyN(f,n-1,f(x)));}
5  }
6  applyN(succ,10,5);
```

- 26 – 28 : appel de $f(x)$
- 29 – 32 : $n - 1$
- 33 : f et 34 : *applyN*
- 35 : appel de *applyN*

```
1    GALLOC
2    JUMP L2
3  L1:
4    PUSH INT 1
5    FETCH 0
6    PUSH PRIM 0
7    CALL 2
8    RETURN
9    PUSH UNIT
10   RETURN
11 L2:
12   PUSH FUN L1
13   GSTORE 0
14   GALLOC
15   JUMP L4
16 L3:
```

```
17   PUSH INT 0
18   FETCH 1
19   PUSH PRIM 4
20   CALL 2
21   JFALSE L5
22   FETCH 2
23   RETURN
24   JUMP L6
25 L5:
26   FETCH 2
27   FETCH 0
28   CALL 1
29   PUSH INT 1
30   FETCH 1
31   PUSH PRIM 1
32   CALL 2
```

```
33   FETCH 0
34   GFETCH 1
35   CALL 3
36   RETURN
37 L6:
38   PUSH UNIT
39   RETURN
40 L4:
41   PUSH FUN L3
42   GSTORE 1
43   PUSH INT 5
44   PUSH INT 10
45   GFETCH 0
46   GFETCH 1
47   CALL 3
48   POP
```

Passage de fonction comme paramètre (2)

```
1  function succ (x) { return (x+1) ;}
2  function mult5(y) { return (5 * y) ;}
3  function compose(f,g,x) {
4    return (f(g(x))) ;
5  }
6  compose(succ,mult5,20);
```

- L5 : définition de *compose*
- L6 : appel de *compose*
  - *mult*5 : `GFETCH 1`
  - *succ* : `GFETCH 0`
  - *compose* : `GFETCH 2`

```
29  L5:
30    FETCH 2
31    FETCH 1
32    CALL 1
33    FETCH 0
34    CALL 1
35    RETURN
36    PUSH UNIT
37    RETURN
38  L6:
39    PUSH FUN L5
40    GSTORE 2
41    PUSH INT 20
42    GFETCH 1
43    GFETCH 0
44    GFETCH 2
45    CALL 3
46    POP
```

Retour d'une valeur fonctionnelle

```
3  function compose(f,g) {
4    function c(x) {return (f(g(x)));}
5    return (c);
6  }
7  let h = compose(succ,mult5) ;
8  h(20);
```

- *succ* : L1 et *mult*5 : L3
- *c* : L7 et *compose* : L8
- déclaration de *h* : L6
- appel de *h* : L9

```
1    GALLOC
2    JUMP L2
3  L1:
4    PUSH INT 1
5    FETCH 0
6    PUSH PRIM 0
7    CALL 2
8    RETURN
9    PUSH UNIT
10   RETURN
11 L2:
12   PUSH FUN L1
13   GSTORE 0
14   GALLOC
15   JUMP L4
16 L3:
```

```
    FETCH 0
    PUSH INT 5
    PUSH PRIM 2
    CALL 2
    RETURN
    PUSH UNIT
    RETURN
L4:
    PUSH FUN L3
    GSTORE 1
    GALLOC
    JUMP L6
L5:
    GALLOC
    JUMP L8
L7:
```

```
    FETCH 0
    FETCH 2
    CALL 1
    FETCH 1
    CALL 1
    RETURN
    PUSH UNIT
    RETURN
L8:
    PUSH FUN L7
    GSTORE 3
    GFETCH 3
    RETURN
    PUSH UNIT
    RETURN
L6:
```

```
49   PUSH FUN L5
50   GSTORE 2
51   GFETCH 1
52   GFETCH 0
53   GFETCH 2
54   CALL 2
55   JUMP L10
56 L9:
57   PUSH INT 20
58   FETCH 0
59   CALL 1
60   POP
61   PUSH UNIT
62   RETURN
63 L10:
64   PUSH FUN L9
65   CALL 1
```

Enrichir la biliothèque d'exécution (1)

paires:

- ajouter un type pour l'allocation :

```
1  typedef struct _pair {
2    value_t car;  /*!< premier e'le'ment de la paire. */
3    value_t cdr;  /*!< second e'le'ment de la paire. */
4    int gc_mark;   /*!< la valeur de la marque (0 ou 1). */
5  } pair_t;
```

- ajouter un cas dans l'union value_data :

```
1    struct _pair    *as_pair;    /*!< si c'est une paire (T_PAIR) */
```

- des fonctions utilitaires :

```
1  int value_is_pair(value_t *value);
2  pair_t * value_pair_get(value_t *value);
3  value_t *value_get_car(value_t *value);
4  value_t *value_get_cdr(value_t *value);
5  void value_set_car(struct _vm * vm, value_t *value, value_t *car);
6  void value_set_cdr(struct _vm * vm, value_t *value, value_t *cdr);
```

Enrichir la biliothèque d'exécution (2)

- définir des primitives : constructeur, accesseurs (car,cdr)

```
1  void do_cons_prim(vm_t *vm, varray_t *stack) {
2   // en sommet de pile on trouve les deux arguments: [car cdr ...]
3   varray_expandn(stack, 1);  // ajout d'une place pour le re'sultat
4   value_fill_nil(varray_top(stack));     // initialiser une paire vide
5   // on place le car et le cdr
6   value_set_car(vm,varray_top(stack), varray_top_at(stack, 1));
7   value_set_cdr(vm,varray_top(stack), varray_top_at(stack, 2));
8   // puis on copie le re'sultat -> [res cdr res ...]
9   varray_set_top_at(stack, 2, varray_top(stack));
10  varray_popn(stack, 2);//on de'pile les 2 e'le'ments -> [res ...]
11 }
12 void do_car_prim(varray_t *stack) {
13  varray_set_top(stack, value_get_car(varray_top(stack)));  }
```

- et les inclure dans execute_prim

```
1  void execute_prim(vm_t * vm, varray_t *stack, int prim, int n) {
2   switch(prim) {
3   case P_CONS:
4     do_cons_prim(vm,stack); break;
5   case P_CAR:
6     do_car_prim(stack); break;
```

Enrichir la biliothèque d'exécution (3)

On cherche à définir une structure de blocs pour pouvoir représenter les n-uplets, les enregistrements et les tableaux.

- possibilité en reprennant la structure `varray_t` existante

```
1  typedef struct _block {
2    int gc_mark; /*!< marque pour le GC */
3    varray_t * content; /*!< tableau de cellules. */
4  } block_t;
```

- ajout du type : `T_BLOCK`
- un cas dans `data_value`
- des fonctions utilitaires : construction et accès
- intégration dans les primitives
- cas particulier pour ces primitives
- impact sur le GC (voir prochain cours)

Assembleur et byte-code

prog1.js (slide 20)

- assembleur : sortie du compilateur `-compile`

```
1    GALLOC
2    JUMP L2
3  L1:
4    PUSH INT 1
5    FETCH 0
6    PUSH PRIM 0
7    CALL 2
8    RETURN
9    PUSH UNIT
10   RETURN
11 L2:
12   PUSH FUN L1
13   GSTORE 0
14   PUSH INT 5
15   GFETCH 0
16   CALL 1
17   POP
```

- format décimal du byte-code

```
1  424242 30 0 4 17 1 1 1 8 0 1 2 0 6 2 7 1 0 7 1 3 3 2 0 1 1 5 5 0 6 1 3
```

- en-tête
  - magic number : 424242 pour identifier le byte-code
  - taille : 30 sur prog1.js.bc
- dernier caractère du fichier : espace (code ascii 32)

Sérialisation du byte-code

numérotation fixe des types, primitives et instructions

fichier `constants.h`:

```
1  /* Constantes
2  pour les opcodes */
3  #define I_LABEL -1
4  #define I_GALLOC 0
5  #define I_PUSH 1
6  #define I_GSTORE 2
7  #define I_POP 3
8  #define I_JUMP 4
9  #define I_GFETCH 5
10 #define I_CALL 6
11 #define I_RETURN 7
12 #define I_FETCH 8
13 #define I_JFALSE 9
14 #define I_STORE 10
```

```
1  /* Constantes
2  pour les types ↩
       */
3  #define T_UNIT 0
4  #define T_INT 1
5  #define T_PRIM 2
6  #define T_FUN 3
7  #define T_BOOL 4
```

```
1  /* Constantes pour
2      les primitives ↩
           */
3  #define P_ADD 0
4  #define P_SUB 1
5  #define P_MUL 2
6  #define P_DIV 3
7  #define P_EQ 4
```

Assembleur et byte-code

```
1 424242 30 0 4 17 1 1 1 8 0 1 2 0 6 2 7 1 0 7 1 3 3 2 0 1 1 5 5 0 6 1 3
```

- 0 : GALLOC
- 4 17 : JUMP de la position L2
- 1 1 1 : PUSH INT 1
- 8 0 : FETCH 0
- 1 2 0 : PUSH PRIM 0
- 6 2 : CALL 2
- 7 : RETURN
- 1 0 7 : PUSH UNIT suivi de RETURN (code mort)
- 1 3 3 : PUSH FUN L1
- 2 0 : GSTORE
- 1 1 5 : PUSH INT 5
- 5 0 : GFETCH 0
- 6 1 : CALL 1

Sérialisation du byte-code

Deux passes :

- calcul des adresses des labels (en fonction de la taille des instructions)
  - `L1:` indice 3 (`GALLOC` et `JUMP L2` prenant 3 cellules)
  - `L2:` : indice 17
- traduction des instructions, des types et des labels en décimal

programme indépendant du compilateur ou de la machine virtuelle, et dépendant de la numérotation des instructions , des types et des primitives de MV_{3I018}.