Gambit Virtual Machine

+

Mapping by C and Native Backends

Gambit Virtual Machine

- GVM is the intermediate code representation of the compiler (Scheme —> GVM —> target)
- A procedure's code is a **set of basic-blocks** with **explicit jump instructions** for local and global control flow
  - Each BB is a **control point** of the program
  - Cases: procedure entry-point, procedure return-point, closure entry-point, or internal control point

Gambit Virtual Machine

- The GVM has a set of general purpose **registers** and a **stack**, both used by the procedure call protocol

- The number of registers of the GVM depends on the backend

- The frontend adjusts its GVM code generation to agree with the requirements of the backend (total nb. registers and nb. of argument passed in registers)

foreach.scm

```
(declare
  (standard-bindings)
  (not safe)
  (inlining-limit 0)
  (not interrupts-enabled)
)

(define (foreach f lst)
  (let loop ((lst lst))
    (if (pair? lst)
        (begin
          (f (car lst))
          (loop (cdr lst)))
        #f)))

(foreach println '(1 2 3))
```

gsc -gvm -cfg foreach.scm

foreach.gvm

```
**** #<primitive foreach#> =
#1 fs=0 entry-point nparams=0 ()                  [] r0=#ret
  global[foreach] = '#<procedure foreach>         [] r0=#ret
  r2 = '(1 2 3)                                   [] r0=#ret r2=#
  r1 = global[println]                            [] r0=#ret r1=# r2=#
  jump fs=0 global[foreach] nargs=2               [] r0=#ret r1=# r2=#

**** #<procedure foreach> =
#1 fs=0 entry-point nparams=2 ()                  [] r0=#ret r1=f r2=lst
  if (##pair? r2) jump fs=0 #3 else #4            [] r0=#ret r1=f r2=lst
#2 fs=8 return-point                              [#ret f lst . . . . .] r1=#
  r2 = (##cdr frame[3])                           [#ret f . . . . . .] r2=#
  r1 = frame[2]                                   [#ret . . . . . . .] r1=# r2=#
  r0 = frame[1]                                   [. . . . . . . . .] r0=#ret r1=# r2=#
  if (##pair? r2) jump fs=0 #3 else #4            [] r0=#ret r1=# r2=#
#3 fs=0                                           [] r0=#ret r1=f r2=lst
  frame[1] = r0                                   [#ret] r1=f r2=lst
  frame[2] = r1                                   [#ret f] r1=f r2=lst
  frame[3] = r2                                   [#ret f lst] r1=f r2=lst
  r1 = (##car r2)                                 [#ret f lst] r1=# r2=lst
  jump fs=8 frame[2] r0=#2 nargs=1                [#ret f lst . . . . .] r0=# r1=# r2=lst
#4 fs=0                                           [] r0=#ret
  r1 = '#f                                        [] r0=#ret r1=#
  jump fs=0 r0                                    [] r1=#
```

content of stack frame

"." = dead slot

gsc -gvm -cfg foreach.scm

foreach.cfg.pdf

#<primitive foreach#>

```
#1 fs=0 entry-point nparams=0 ()
  global[foreach] = '#<procedure foreach>
  r2 = '(1 2 3)
  r1 = global[println]
  jump fs=0 global[foreach] nargs=2
```

#<procedure foreach>

```
#1 fs=0 entry-point nparams=2 ()
  if (##pair? r2) jump fs=0 #3 else #4
```

```
#2 fs=8 return-point
  r2 = (##cdr frame[3])
  r1 = frame[2]
  r0 = frame[1]
  if (##pair? r2) jump fs=0 #3 else #4
```

entry frame size

exit frame size

```
#3 fs=0
  frame[1] = r0
  frame[2] = r1
  frame[3] = r2
  r1 = (##car r2)
  jump fs=8 frame[2] r0=#2 nargs=1
```

```
#4 fs=0
  r1 = '#f
  jump fs=0 r0
```

GVM Example with Polling

```
(define (foreach f lst)
  (let loop ((lst lst))
    (if (pair? lst)
        (begin
          (f (car lst))
          (loop (cdr lst)))
        #f)))
```

Call protocol of **foreach** and **loop**

r0 = *return address*
r1 = f
r2 = lst

Call protocol of **f**

r0 = *return address*
r1 = *arg1*

GVM basic blocks of **foreach**

```
#1 fs=0 entry-point nparams=2 ()
   jump fs=0 #3

#2 fs=3 return-point
   r2 = (cdr frame[3])
   r1 = frame[2]
   r0 = frame[1]
   jump/poll fs=0 #3

#3 fs=0
   if (pair? r2) jump fs=0 #4 else #5

#4 fs=0
   frame[1] = r0
   frame[2] = r1
   frame[3] = r2
   r1 = (car r2)
   r0 = #2
   jump fs=3 frame[2] nargs=1

#5 fs=0
   r1 = '#f
   jump fs=0 r0
```

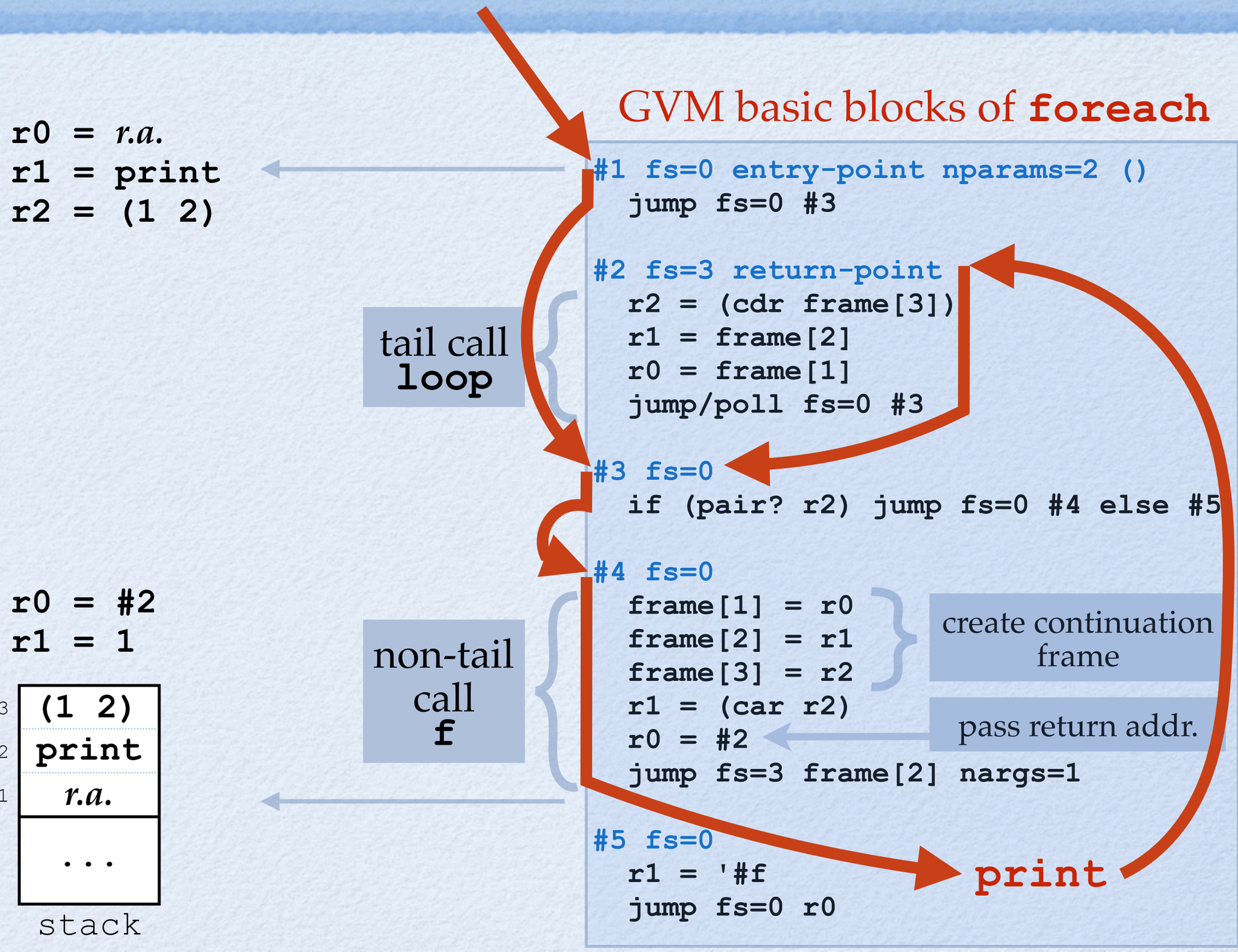
tail call **loop**

non-tail call **f**

create continuation frame

pass return addr.

Trace of `(foreach print '(1 2))`



Trace of `(foreach print '(1 2))`

```
r0 = r.a.
r1 = print
r2 = (1 2)
```

GVM basic blocks of **`foreach`**

```
#1 fs=0 entry-point nparams=2 ()
   jump fs=0 #3

#2 fs=3 return-point
   r2 = (cdr frame[3])
   r1 = frame[2]
   r0 = frame[1]
   jump/poll fs=0 #3

#3 fs=0
   if (pair? r2) jump fs=0 #4 else #5

#4 fs=0
   frame[1] = r0
   frame[2] = r1
   frame[3] = r2
   r1 = (car r2)
   r0 = #2
   jump fs=3 frame[2] nargs=1

#5 fs=0
   r1 = '#f
   jump fs=0 r0
```

tail call **`loop`**

create continuation frame

pass return addr.

`print`

`jump/poll` instruction performs a check for interrupts (UI events, etc) in addition to the actual jump

The frontend guarantees that a **bounded number of instructions are executed between polling operations**

Stack

Object representation

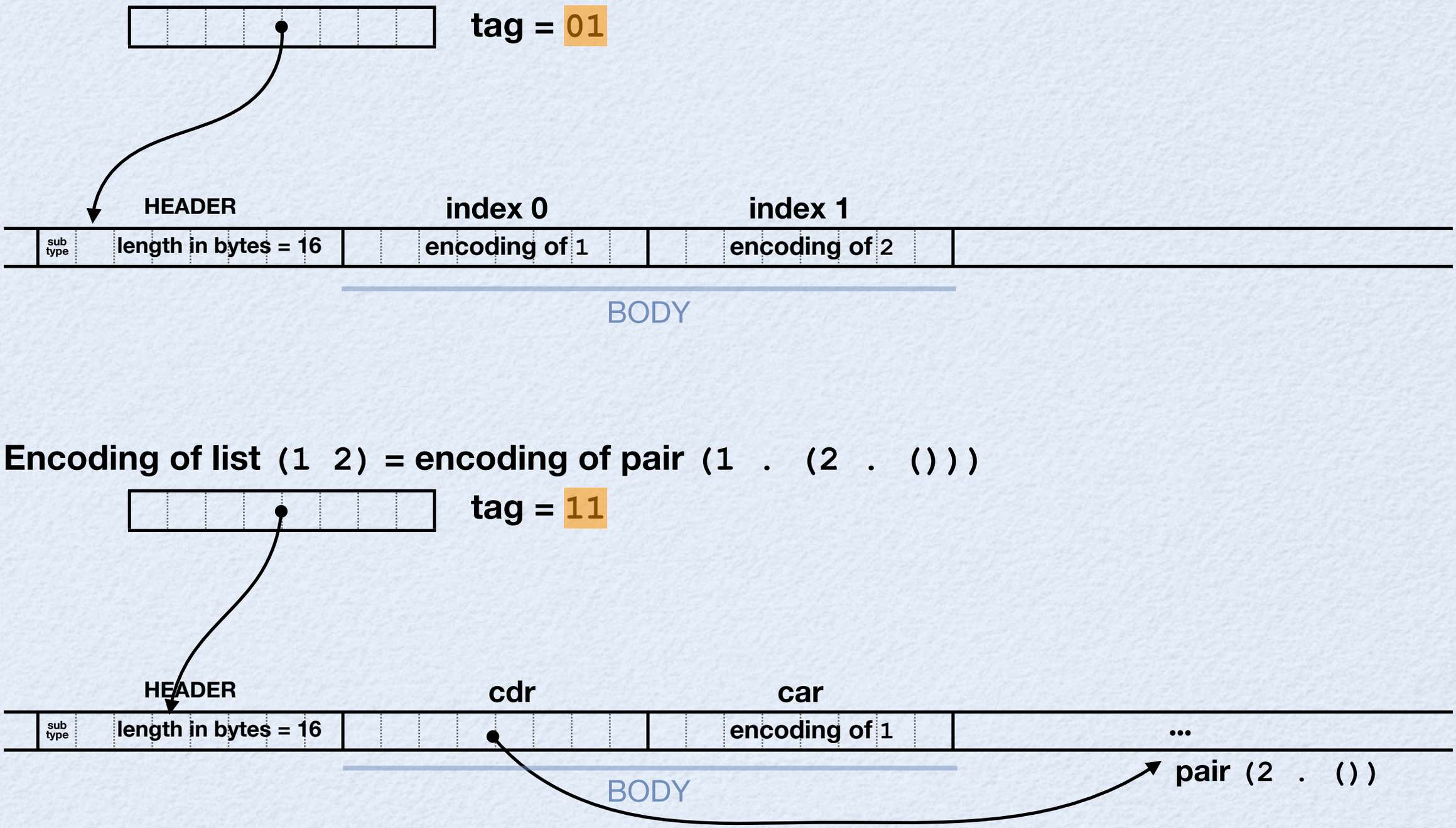
- How are values (objects) represented?
- Because Scheme is a dynamically typed language, the representation of Scheme objects carries type information at run time
- Frequently used types need to be handled efficiently:
  - fixnums (small exact integers fitting in a word)
  - special objects (`#f`, `#t`, `()`, ...)
  - pairs
- All objects are encoded with 1 word, plus more when it is a memory allocated object

Object encoding

```
> (define (binrepr obj) (number->string (##object->encoding obj) 2))

> (map binrepr '(0 1 2 3 4 5 6))
("0" "100" "1000" "1100" "10000" "10100" "11000")  ;; fixnum tag = 00

> (map binrepr '(#f #t () #!void #!eof #\newline #\A))
("11111111111111111111111111111110" ;; #f -> -2       special tag = 10
 "11111111111111111111111111111010" ;; #t -> -6
 "11111111111111111111111111110110" ;; () -> -10
 "11111111111111111111111111011110" ;; #!void -> -18
 "11111111111111111111111100011110" ;; #!eof  -> -14
 "101010"                           ;; #\newline -> 42
 "10000010")                        ;; #\A -> 262

> (binrepr (vector 1 2))
"1111011111001101011101000101000101" ;; address of vector      tag = 01

> (binrepr (list 1 2))
"1111011111001101011111000001101011" ;; address of first pair  tag = 11

> (##encoding->object #b1111011111001101011111000001101011) ;; danger!
(1 2)
```

Memory Allocated Objects

Encoding of vector #(1 2)



Encoding of list (1 2) = encoding of pair (1 . (2 . ()))

Subtype Field

- The 8 bit subtype field contains:
  - 5 bit type information (vector = 0, pair = 1, …)
  - 3 bits to encode GC information and memory allocation method:
    - **permanent**: address of object in memory never changes and memory is never reclaimed
    - **still**: address of object in memory never changes
    - **movable**: GC may move the object in memory

gambit.h

```
#define ___SB 5
#define ___sVECTOR        0
#define ___sPAIR          1
#define ___sRATNUM        2
#define ___sCPXNUM        3
#define ___sSTRUCTURE     4
#define ___sBOXVALUES     5
#define ___sMEROON        6
#define ___sJAZZ          7
#define ___sSYMBOL        8
#define ___sKEYWORD       9
#define ___sFRAME         10
#define ___sCONTINUATION  11
#define ___sPROMISE       12
#define ___sWEAK          13
#define ___sPROCEDURE     14
#define ___sRETURN        15
#define ___sFOREIGN       18
#define ___sSTRING        19
#define ___sS8VECTOR      20
#define ___sU8VECTOR      21
#define ___sS16VECTOR     22
#define ___sU16VECTOR     23
#define ___sS32VECTOR     24
#define ___sU32VECTOR     25
#define ___sF32VECTOR     26
#define ___sS64VECTOR     27
#define ___sU64VECTOR     28
#define ___sF64VECTOR     29
#define ___sFLONUM        30
#define ___sBIGNUM        31
```

gambit.h

```
/*
 * Type tag assignment.
 *
 * Type tags are located in the lower 2 bits of a ___SCMOBJ.
 *
 * ___TB = number of tag bits
 * ___tFIXNUM   = tag for fixnums (small integers), must be 0
 * ___tSPECIAL  = tag for other immediates (#f, #t, (), ...)
 * ___tMEM1     = tag #1 for memory allocated objects
 * ___tMEM2     = tag #2 for memory allocated objects
 * ___tSUBTYPED = ___tMEM1
 * ___tPAIR     = ___tMEM1 or ___tMEM2
 */

#define ___TB 2
#define ___tFIXNUM   0
#define ___tMEM2     3

#ifdef ___USE_EVEN_TAG_FOR_SUBTYPED
#define ___tSPECIAL  1
#define ___tMEM1     2
#else
#define ___tSPECIAL  2
#define ___tMEM1     1
#endif

#define ___tSUBTYPED ___tMEM1

#ifdef ___USE_SAME_TAG_FOR_PAIRS_AND_SUBTYPED
#define ___tPAIR ___tMEM1
#else
#define ___tPAIR ___tMEM2
#endif

#define ___MEM_ALLOCATED(obj) ((obj)&___tMEM1)
```

gambit.h

```
#define ___FIX(x) ((___WORD)((___UWORD)x)<<___TB)
#define ___INT(x) ((x)>>___TB)

#define ___FIXNUMP(x) (((x)&3) == ___tFIXNUM)
#define ___FIXADD(x,y) ((___WORD)((x)+(y)))
#define ___FIXSUB(x,y) ((___WORD)((x)-(y)))
#define ___FIXMUL(x,y) ((___WORD)((x)*___INT(y)))
...

#define ___PAIR_SIZE 2
#define ___PAIR_CDR 0
#define ___PAIR_CAR 1

#define ___CAR_FIELD(obj) ___PAIR_TO_BODY(obj)[___PAIR_CAR]
#define ___CDR_FIELD(obj) ___PAIR_TO_BODY(obj)[___PAIR_CDR]

#define ___ALLOC_PAIR_EXPR(x,y)(___BEGIN_ALLOC_PAIR(), \
___ADD_PAIR_ELEM(___PAIR_CAR,x), \
___ADD_PAIR_ELEM(___PAIR_CDR,y), \
___END_ALLOC_PAIR())

#define ___CONS(x,y)(___ALLOC_PAIR_EXPR(x,y),___GET_PAIR())

#define ___SETCAR(obj,car)___CAR_FIELD(obj)=car;
#define ___SETCDR(obj,cdr)___CDR_FIELD(obj)=cdr;
#define ___CAR(obj)___CAR_FIELD(obj)
#define ___CDR(obj)___CDR_FIELD(obj)
...
```

Low Cost for Tagging

- Many fixnum operations don't have to add or remove the type tag, or for only one operand
- Access to fields (`car`/`cdr`/`vector-ref`/...) can combine in a single machine instruction the untagging with the offset of the field
- That is a simple optimization of the C compiler, or the Gambit native backend
- Let's show this through an example of run time code generation with **`gsc`**

Run Time Code Generation

x86-asm.scm

```
(include "~~lib/_asm#.scm") ;; Import compiler fns
(include "~~lib/_x86#.scm")
(include "~~lib/_codegen#.scm")

;; Convert a u8vector containing machine code into a
;; Scheme procedure taking 0 to 3 arguments.  Calling
;; the Scheme procedure will execute the machine code
;; using the C calling convention.

(define (u8vector->procedure code)
  (let ((mcb (##make-machine-code-block code)))
    (lambda (#!optional (arg1 0) (arg2 0) (arg3 0))
      (##machine-code-block-exec mcb arg1 arg2 arg3))))

;; Create a new code generation context.  The format of
;; the resulting assembly code listing can also be
;; specified, either 'nasm, 'gnu, or #f (no listing,
;; which is the default).

(define (make-cgc #!optional (format #f))
  (let ((cgc (make-codegen-context)))
    (asm-init-code-block cgc 0 endianness)
    (codegen-context-listing-format-set! cgc format)
    (x86-arch-set! cgc arch)
    cgc))
```

Run Time Code Generation

```
(define arch 'x86-64)
(define endianness 'le)

(define (asm gen #!optional (format 'gnu))
  (let ((cgc (make-cgc format)))

    (gen cgc)

    (let ((code (asm-assemble-to-u8vector cgc)))

      (if format
          (asm-display-listing cgc
                               (current-error-port)
                               #t)) ;; hex code too!

      (u8vector->procedure code))))
```

Run Time Code Generation

```
(define f5
  (asm
   (lambda (cgc)
     (x86-mov cgc (x86-rax) (x86-imm-int (* 5 4)))
     (x86-ret cgc)  ;; return Scheme value 5 in rax
     )))

(define nth
  (asm
   (lambda (cgc)

     (define loop (asm-make-label cgc 'loop))
     (define test (asm-make-label cgc 'test))
     (define lst  (x86-rdi))
     (define i    (x86-rsi))
     (define (getcar x) (x86-mem 13 x))
     (define (getcdr x) (x86-mem 5  x))

     (x86-cmp   cgc i (x86-imm-int 0))
     (x86-jmp   cgc test)

     (x86-label cgc loop)
     (x86-mov   cgc lst (getcdr lst))
     (x86-sub   cgc i (x86-imm-int 4))

     (x86-label cgc test)
     (x86-jne   cgc loop)

     (x86-mov   cgc (x86-rax) (getcar lst))
     (x86-ret   cgc)  ;; return result in rax

     )))

(pp (nth '(100 101 102 103 104 105 106 107) (f5)))
```

```
x86 C calling convention:

 x86-32  x86-64
 0(esp)  0(rsp)  return addr
 4(esp)     rdi  arg1
 8(esp)     rsi  arg2
12(esp)     rdx  arg3
    eax     rax  fn result
```

```
(define (nth lst n)
  (if (fx= n 0)
      (car lst)
      (nth (cdr lst)
           (fx- n 1))))
```

Run Time Code Generation

```
% gsc -i x86-asm.scm
==> creating f5:
000000
000000 48c7c014000000
000007 c3
==> creating nth:
000000
000000 4883fe00
000004 eb08
000006
000006 488b7f05
00000a 4883ee04
00000e
00000e 75f6
000010 488b470d
000014 c3
105
```

```
        .code64
        movq    $20,%rax
        ret
```

```
        .code64
        cmpq    $0,%rsi
        jmp     test
loop:
        movq    5(%rdi),%rdi
        subq    $4,%rsi
test:
        jne     loop
        movq    13(%rdi),%rax
        ret
```