Native backends

Laurent Huberdeau

Abdelhakim Qbaich

Why?

- C imposes code generation constraints
  - Affects performance
    - Tail calls
    - Continuations
- Possible dynamic code generation for specific architectures
  - Useful for hardware-oriented or other specific tasks

How?

- Gambit uses the Gambit Virtual Machine
  - Intermediate representation
  - High-level abstraction
- Interoperability with the C backend
  - No need to rewrite the entirety of the runtime system
    - Garbage collector
    - OS interface and library procedures
    - Exception handlers
  - Allows incremental progress
    - Automatic fallback to the C backend

Supported architectures

- By the compiler
  - *i386*, *x32*, *amd64*, *armel*, *armhf*, *arm64*, *riscv32*, *riscv64*, and many others
- By the native backend
  - *i386*, *amd64*, *armhf*, *riscv32*, *riscv64*

Internals

- GVM Registers
  - 5 general purpose
    - *R0*: return address of current procedure
    - *R1*: return value of procedures
    - *R4*: self register for closure calls
  - 3 special purpose
    - Processor state pointer
    - Stack pointer
    - Heap pointer
  - 0+ temporary

GVM	x86 (32-bit)	x86 (64-bit)	ARM	RISC-V
R0	edi	rdi	r0	s2
R1	eax	rax	r1	s3
R2	ebx	rbx	r2	s4
R3	edx	rdx	r3	s5
R4	esi	rsi	r4	s6
PS	ecx	rcx	r5	s10
SP	esp	rsp	r13	sp
HP	ebp	rbp	r6	s11
TMP0			r7	s7
TMP1				s8

Internals

- GVM instructions
  - *label*: identifier of a basic block
    - First instruction of every basic block
    - Indicates current size of stack frame
  - *jump*: unconditional branches
    - Verification by callee of number of arguments
    - Support for optional and rest parameters
    - Destination: closure, an entry point of a procedure, or a return point
  - *ifjump*, *switch*: conditional branches (*if*, *cond*, *case*)
  - *copy*: assignment of a value
    - From the source operand to the destination operand
  - *close*: closure creation
    - Object contains: reference to the entry point and value of its free variables
  - *apply*: execution of a primitive
    - These primitives are inlined

Internals

```
(define (add-if-pos a b)
  (if (and (##fxpositive? a)
           (##fxpositive? b))
      (##fx+ a b)))

(pp (add-if-pos 4 2))
```

#1 fs=0 entry-point nparams=0 ()	[]	r0=#ret
global[add-if-pos] = '#<procedure add-if-pos>	[]	r0=#ret
frame[1] = r0	[#ret]	
r2 = '2	[#ret]	r2=#
r1 = '4	[#ret]	r1=# r2=#
jump/poll fs=4 #2	[#ret . . .]	r1=# r2=#
#2 fs=4	[#ret . . .]	r1=# r2=#
jump fs=4 global[add-if-pos] r0=#3 nargs=2	[#ret . . .]	r0=# r1=# r2=#
#3 fs=4 return-point	[#ret . . .]	r1=#
r0 = frame[1]	[. . . .]	r0=#ret r1=#
jump/poll fs=4 #4	[. . . .]	r0=#ret r1=#
#4 fs=4	[. . . .]	r0=#ret r1=#
jump fs=0 global[pp] nargs=1	[]	r0=#ret r1=#

```
_proc_test#_1:
    cmpq    $0x0,0x68(%rcx)
    jne     narg-error
    movq    $_proc_add-if-pos_1,%r15
    movq    $&global[add-if-pos],%r13
    movq    %r15,(%r13)
    movq    %rdi,-0x8(%rsp)
    movq    $0x8,%rbx
    movq    $0x10,%rax
    addq    $-0x20,%rsp
    cmpq    (%rcx),%rsp
    jl      call-poll-handler2

_proc_test#_3:
    movq    0x18(%rsp),%rdi
    cmpq    (%rcx),%rsp
    jl      call-poll-handler5
```

```
_proc_test#_2:
    movq    $_proc_test#_3,%rdi
    movq    $0x2,0x68(%rcx)
    movq    $&global[add-if-pos],%r15
    movq    (%r15),%r15
    jmp     *%r15

_proc_test#_4:
    addq    $0x20,%rsp
    movq    $0x1,0x68(%rcx)
    movq    $&global[pp],%r15
    movq    (%r15),%r15
    jmp     *%r15
```

```
#1 fs=0 entry-point nparams=2 ()          [] r0=#ret r1=a r2=b
    if (##fxpositive? r1) jump fs=0 #2 else #4  [] r0=#ret r1=a r2=b
#2 fs=0                                   [] r0=#ret r1=a r2=b
    if (##fxpositive? r2) jump fs=0 #3 else #4  [] r0=#ret r1=a r2=b
#3 fs=0                                   [] r0=#ret r1=a r2=b
    r1 = (##fx+ r1 r2)                     [] r0=#ret r1=#
    jump fs=0 r0                           [] r1=#
#4 fs=0                                   [] r0=#ret
    r1 = '#!void                           [] r0=#ret r1=#
    jump fs=0 r0                           [] r1=#
```

```
_proc_add-if-pos_1:
    ld      x24, 112(x26)
    addiw   x22, x0, 2
    bne     x24, x22, narg-error
    bge     x0, x19, _proc_add-if-pos_4
_proc_add-if-pos_2:
    bge     x0, x20, _proc_add-if-pos_4
```

```
_proc_add-if-pos_3:
    add     x19, x19, x20
    jalr    x0, x18, 0
_proc_add-if-pos_4:
    addiw   x19, x0, -18
    jalr    x0, x18, 0
```

RISC-V

- Open-source hardware instruction set architecture
- Based on the reduced instruction set computer principles
  - Small set of simple instructions
  - Typically fewer cycles per instruction
  - Load/store architecture
- Started as academic research at UC Berkeley, funded by DARPA
- Managed by The RISC-V Foundation

Adding a new architecture

- Instruction encoder based on the ISA
  - Instructions and pseudo instructions
  - Pseudo operations (assembler directives)
- Adding the new architecture to the backend
  - Requires: helpers, abstractions, and a lot of primitives
    - In the future, very few primitives and much more abstractions
- Updating parts of the backend for proper support
  - Including trampoline and code generation fixup
- Ideally, having Gambit compile on the target platform
  - Might require specifying in the header: endianness, size, alignment...

```
(define x86-prim-##type
  (const-nargs-prim 1 0 '((reg mem))
    (lambda (cgc result-action args arg1)
      (let ((x86-arg1 (make-x86-opnd arg1)))
        (x86-and cgc x86-arg1 (x86-imm-int type-tag-mask))
        (x86-shl cgc x86-arg1 (x86-imm-int type-tag-bits))
        (am-return-opnd cgc result-action arg1)))))
```

```
(define (x86-prim-field-set index)
  (const-nargs-prim 2 0 '((reg))
    (lambda (cgc result-action args arg1 arg2)
      (let* ((width (get-word-width cgc))
             (offset (+ (body-offset 'subtyped width) (* index width))))
        (am-mov cgc (mem-opnd arg1 offset) arg2)
        (am-return-opnd cgc result-action arg1)))))
```

Benchmarks

- Ack
  - C backend

    Time (mean ± σ):

    3.910 s ± 0.066 s

    Range (min … max):

    3.841 s … 4.022 s
  - x86-64 backend

    Time (mean ± σ):

    3.145 s ± 0.036 s

    Range (min … max):

    3.092 s … 3.193 s
- Sum

  C backend

  Time (mean ± σ):

  2.899 s ± 0.031 s

  Range (min … max):

  2.867 s … 2.959 s

  x86-64 backend

  Time (mean ± σ):

  4.140 s ± 0.156 s

  Range (min … max):

  3.959 s … 4.458 s

Current state

- 56% (218 out of 387) of the primitives implemented for x86, mainly
  - Fixnums
  - Type checks
  - Memory-related (allocation, reference, setting)
- What’s missing?
  - Flonums
  - Bignums
  - Miscellaneous primitives

- ##force
- ##make-will
- ##processor
- ##first-argument
- ##global-var-ref
- ##global-var-set!
- ##global-var-primitive-ref
- ##global-var-primitive-set!
- ##current-vm
- ##current-thread
- ##current-processor
- ##current-processor-id
- ##gc-hash-table-ref
- ##gc-hash-table-set!
- ##gc-hash-table-find!
- ##gc-hash-table-union!
- ##gc-hash-table-rehash!
- ##c-code
- ##check-heap-limit
- ##primitive-lock!
- ##primitive-unlock!
- ##primitive-trylock!
- ##quasi-cons
- ##quasi-list
- ##quasi-vector
- ##vector-inc!
- ##vector-cas!
- ##unchecked-structure-cas!
- ##peek-char0?
- ##peek-char1?
- ##read-char0?
- ##read-char1?
- ##write-char1?
- ##write-char2?

Current state

- 24% (93 out of 387) of the primitives implemented for ARM and RISC-V
  - Still fully functional thanks to the bridge
- Writing the many primitives in assembly is tedious
  - Need for more generalization
    - For example, using a common mapping of a logical shift to the right to :
      - x86’s SHR
      - ARM’s LSR
      - RISC-V’s SRL

Future work

- Refactoring to reduce duplication among the 3 architectures and to simplify addition of more
- Adding the rest of the primitives
- Adding support for AArch64 (64-bit ARM)
- Consider implementations based on extensions like x86’s AVX2, when appropriate
- Simpler setup to debug the execution for the different architectures
- Automatic generation of the instruction encoder code from specification, as a separate module

Quick Demo?