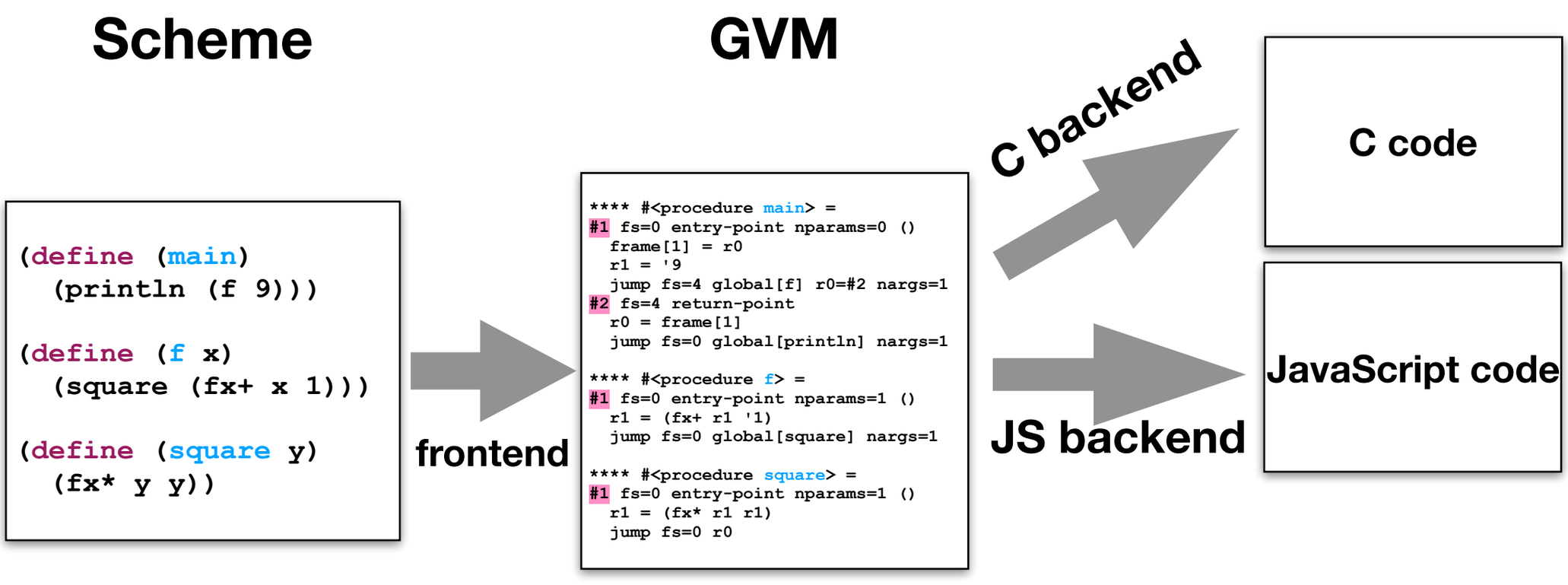
Tail Calling Between Code Generated by C and Native Backends

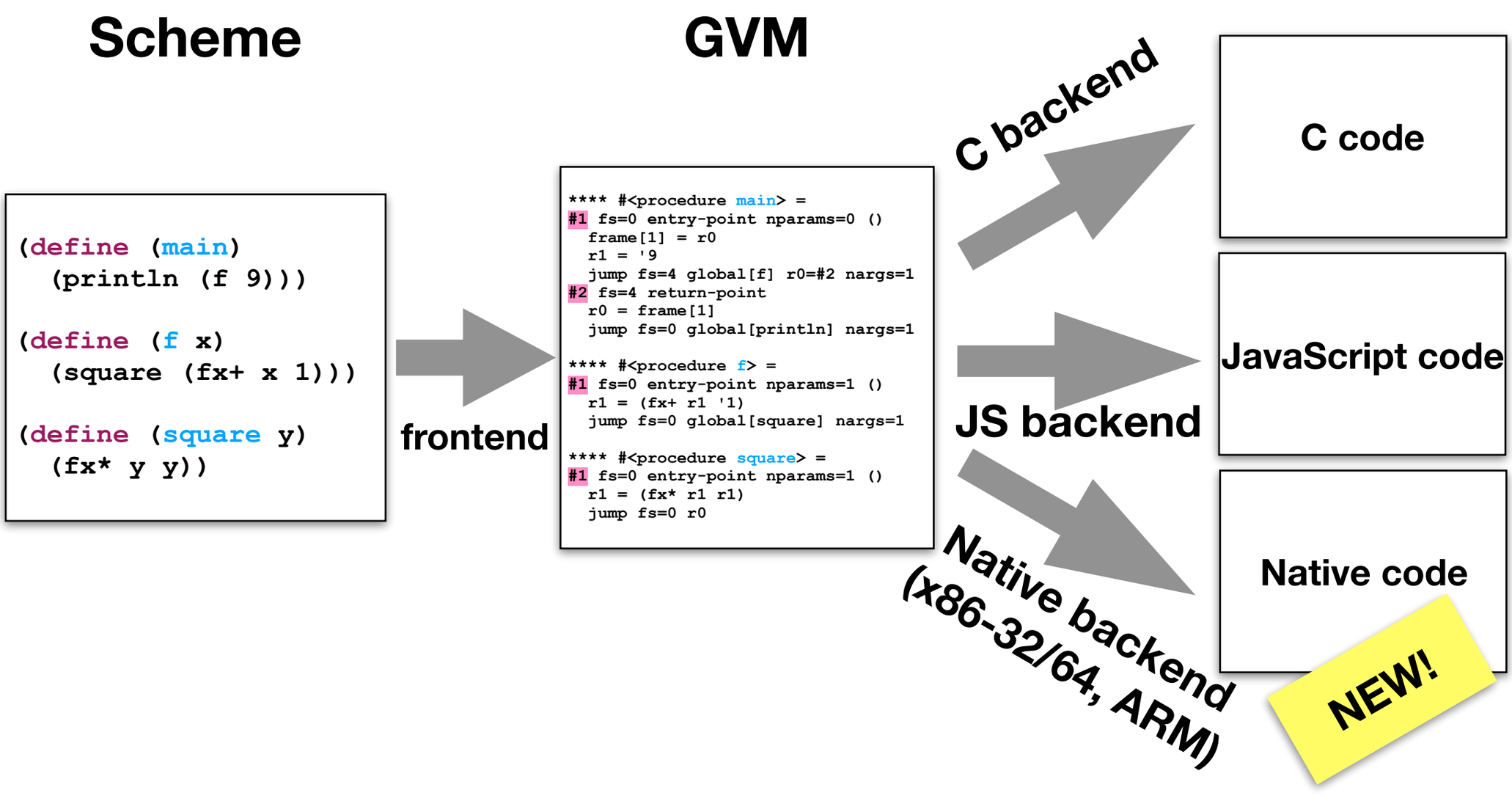
Laurent Huberdeau / Marc Feeley

Université de Montréal

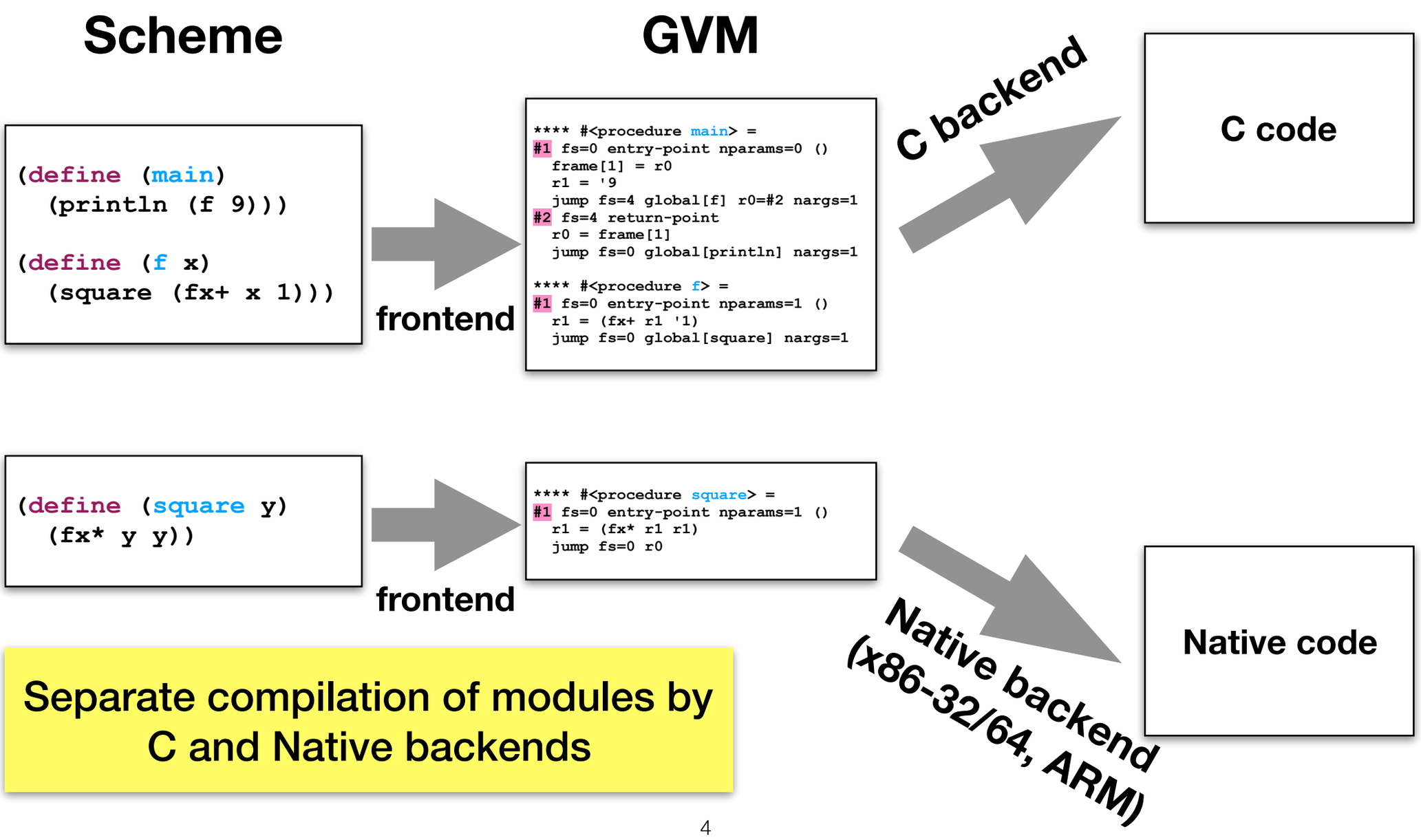
Gambit Compiler Pipeline



Gambit Compiler Pipeline

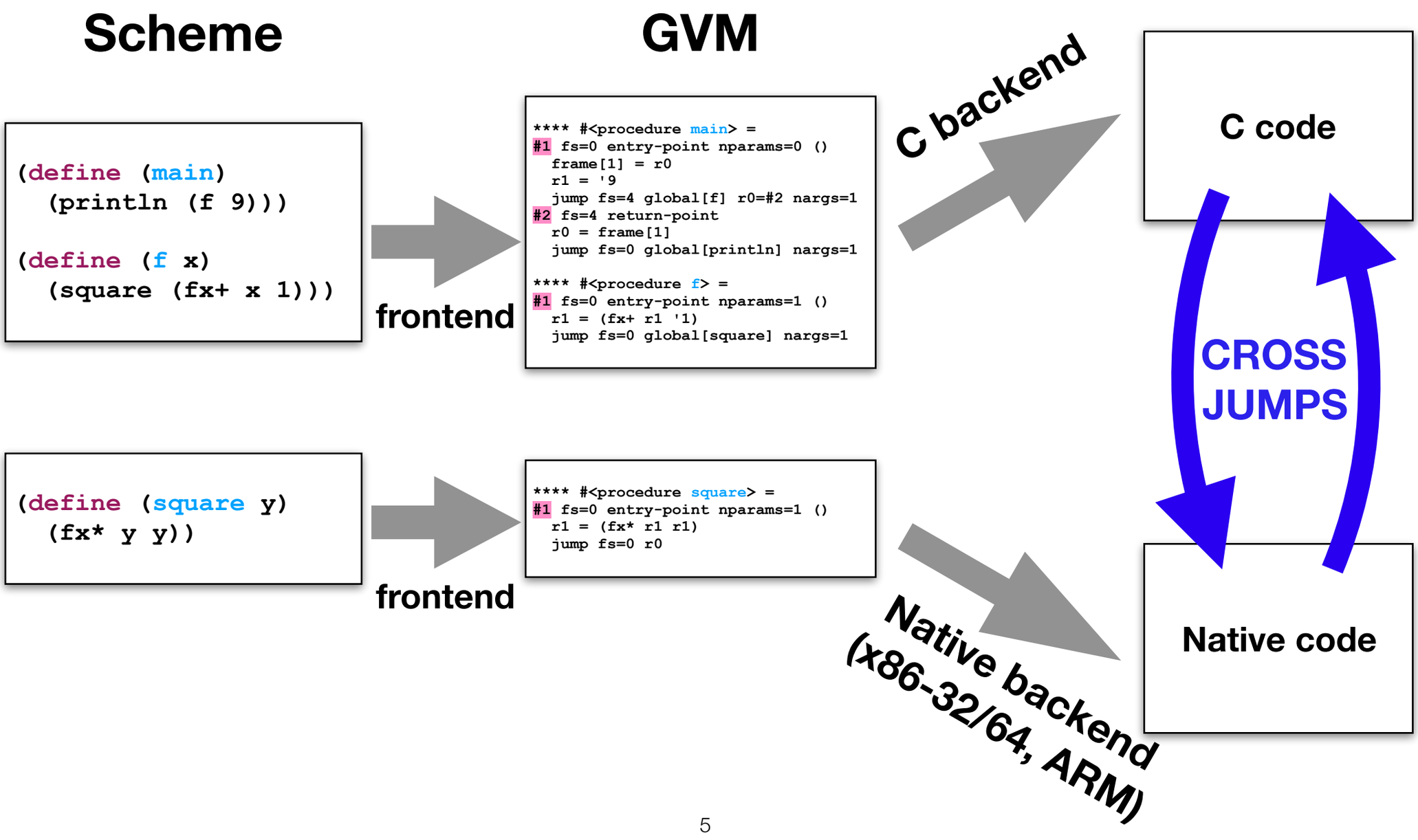


C/Native Interoperability

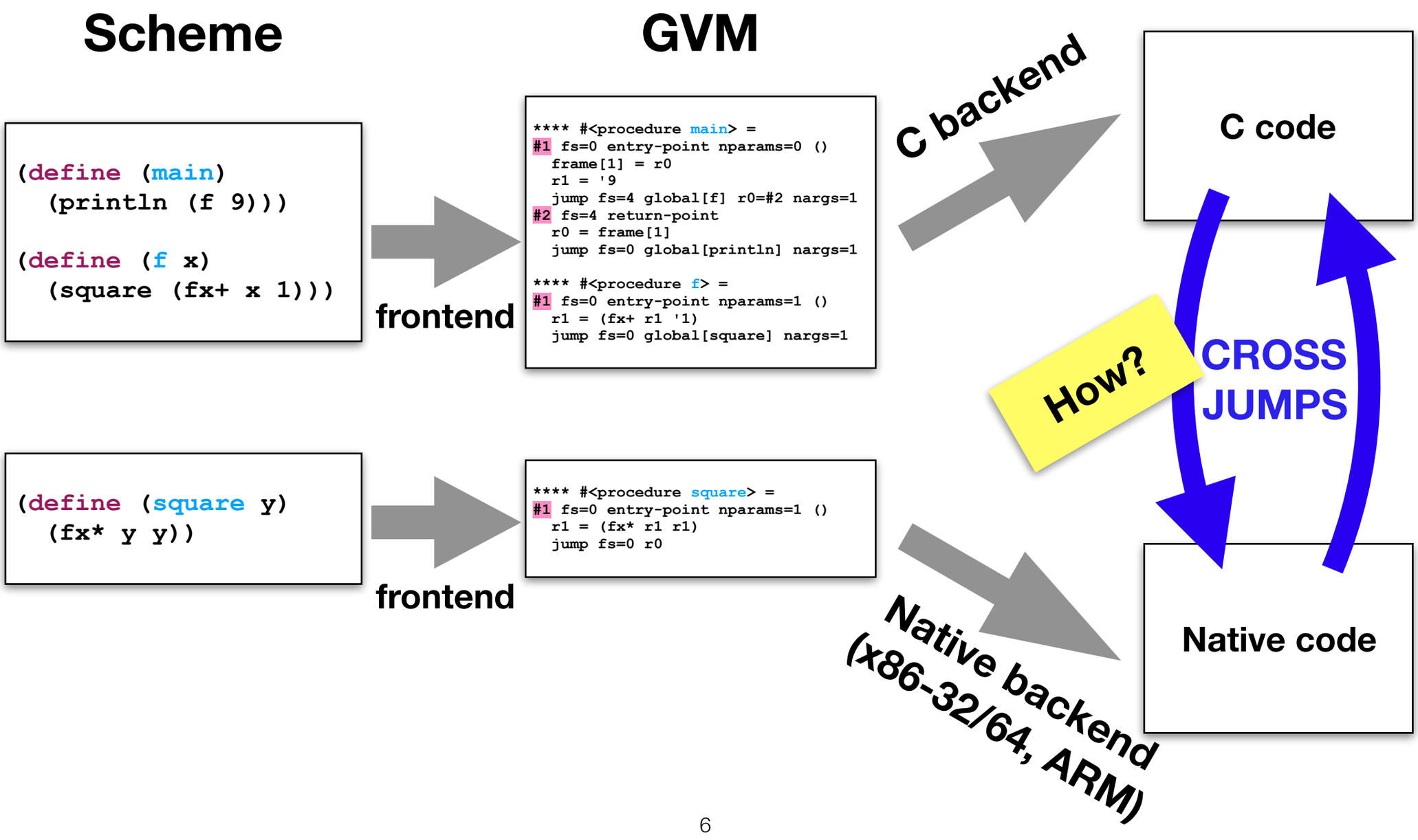


Separate compilation of modules by C and Native backends

C/Native Interoperability



C/Native Interoperability



Why?

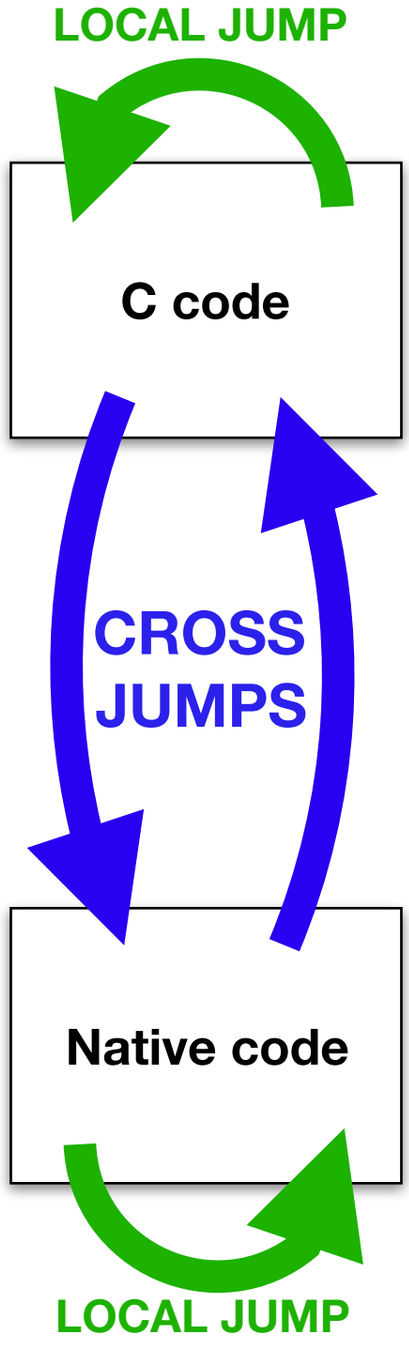
- Reuse runtime library of C backend
  - Complex C functions (garbage collector, OS interface, …)
  - Complex Scheme library procedures (**`eval`**, bignums, …)
  - Complex I/O procedures (**`pretty-print`**, **`read`**, …)
  - Simple primitives, like **`##cons`** and **`##flsqrt`** that are not yet inlined by Native backend (useful during development of backend)
- Mix modules compiled by C and Native backends and compile performance critical parts with Native backend

Desiderata

1. **Cross jumps are transparent** (don't know ahead of time if a jump is going to be a cross jump or a local jump)

2. **Native code jumps are plain *branch-to-address* instructions** (no overhead on native local jumps due to support for cross jumps)

3. **Cost of cross jumps is similar to C backend local jumps** (which are performed with a trampoline)



C Backend

- Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, …)

```
typedef struct processor_state_struct {
  WORD r0, r1, r2, r3, r4;  /* registers   */
  WORD *fp;                 /* frame ptr   */
  WORD *hp;                 /* heap ptr    */
  WORD *pc;                 /* program ctr */
  …
} processor_state_struct;
```

- Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```
typedef struct label_struct {
  WORD HEADER;
  WORD entry;
  WORD unused;
  void (*host)(processor_state_struct *ps);
} label_struct;
```

C Backend

- Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, …)

```
typedef struct processor_state_struct {
  WORD r0, r1, r2, r3, r4;  /* registers   */
  WORD *fp;                 /* frame ptr   */
  WORD *hp;                 /* heap ptr    */
  WORD *pc;                 /* program ctr */
  …
} processor_state_struct;
```

`pc` points to the currently executing control point

- Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```
typedef struct label_struct {
  WORD HEADER;
  WORD entry;
  WORD unused;
  void (*host)(processor_state_struct *ps);
} label_struct;
```

C Backend

- Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, …)

```
typedef struct processor_state_struct {
  WORD r0, r1, r2, r3, r4;  /* registers   */
  WORD *fp;                 /* frame ptr   */
  WORD *hp;                 /* heap ptr    */
  WORD *pc;                 /* program ctr */
  …
} processor_state_struct;
```

`pc` points to the currently executing control point

- Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```
typedef struct label_struct {
  WORD HEADER;
  WORD entry;
  WORD unused;
  void (*host)(processor_state_struct *ps);
} label_struct;
```

The `host` function is the C function containing this control point

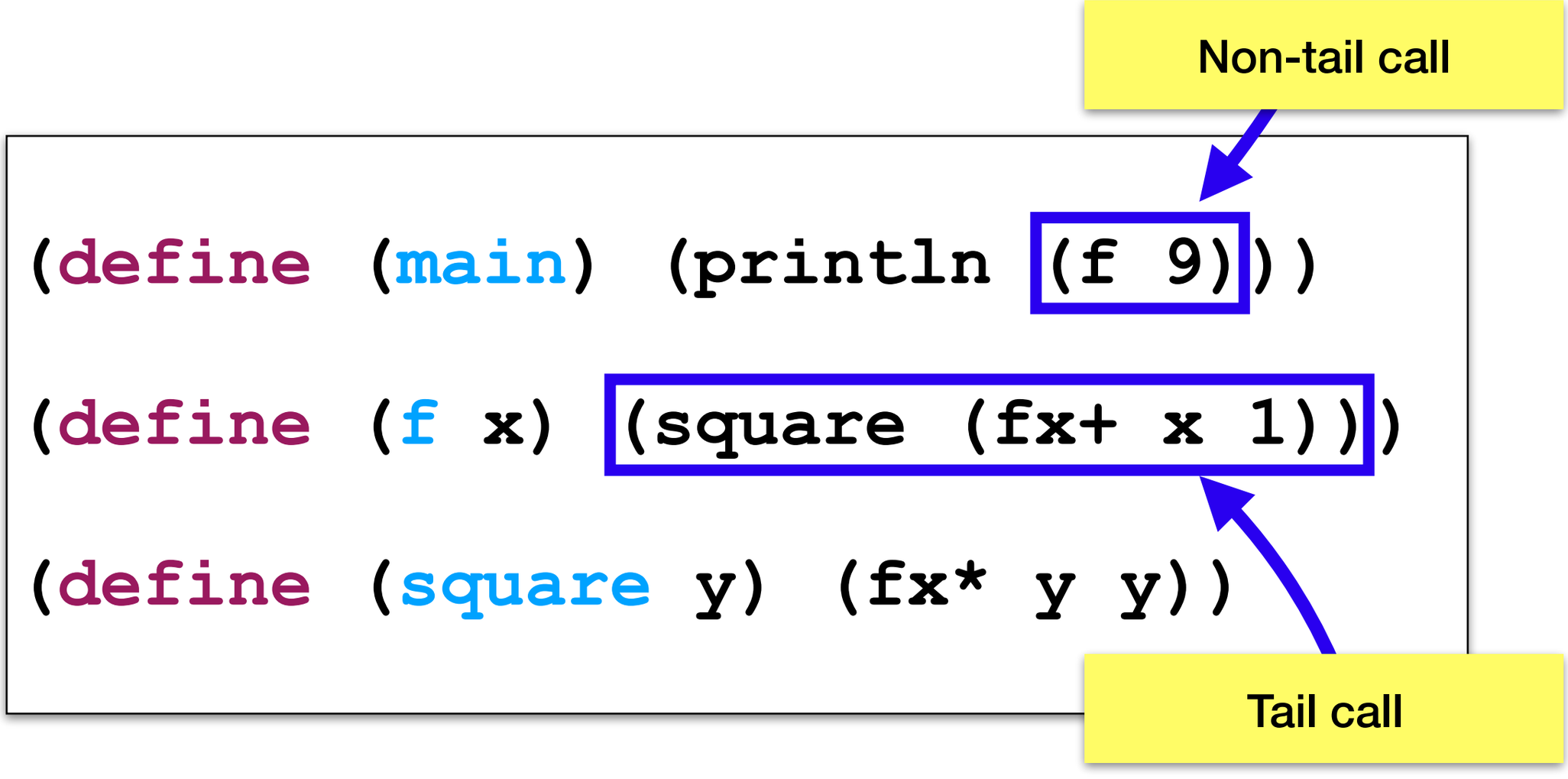
Example

```
(define (main) (println (f 9)))

(define (f x) (square (fx+ x 1)))

(define (square y) (fx* y y))
```

Example


Non-tail call
(define (main) (println (f 9)))
(define (f x) (square (fx+ x 1)))
(define (square y) (fx* y y))
Tail call

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                 /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];     /* tagged return address */
           ps->pc = glo_f;                  /* non-tail call f        */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                     /* add 1 to arg      */
           ps->pc = glo_square;             /* tail call square */
           return;
  }
}
```

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);         /* square arg      */
           ps->pc = ps->r0;                 /* return to main */
           return;
  }
}
```

C host functions

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                  /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];      /* tagged return address */
           ps->pc = glo_f;                   /* non-tail call f        */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                      /* add 1 to arg      */
           ps->pc = glo_square;              /* tail call square */
           return;
  }
}
```

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);          /* square arg      */
           ps->pc = ps->r0;                  /* return to main */
           return;
  }
}
```

Basic Blocks
(first-class control points)

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                  /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];      /* tagged return address */
           ps->pc = glo_f;                   /* non-tail call f        */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

```
ps->r1 = 9 << 2;
ps->r0 = 1+(WORD)&labels[1];
ps->pc = glo_f;
```

```
(define (main) (println (f 9)))

(define (f x) (square (fx+ x 1)))
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                  /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];      /* tagged return address */
           ps->pc = glo_f;                   /*
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

Return address for call to `f` refers to second basic block

```
ps->r1 = 9 << 2;
ps->r0 = 1+(WORD)&labels[1];
ps->pc = glo_f;
```

```
(define (main) (println (f 9)))

(define (f x) (square (fx+ x 1)))
```

Control Points

```
label_struct labels[] = {
   { …, …, …, host_main    }   /* 0 */
,  { …, …, …, host_main    }   /* 1 */
,  { …, …, …, host_f       }   /* 2 */
,  { …, …, …, host_square  }   /* 3 */
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

Control Points

```
label_struct labels[] = {
   { …, …, …, host_main    }   /* 0 */
,  { …, …, …, host_main    }   /* 1 */
,  { …, …, …, host_f       }   /* 2 */
,  { …, …, …, host_square  }   /* 3 */
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

Procedures and return addresses are **tagged**

Return addresses are first-class objects in Gambit Scheme
(allows continuation inspection in Scheme and simplifies GC)

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

```
label_struct labels[] = {
   { …, …, …, host_main    }
 , { …, …, …, host_main    }
 , { …, …, …, host_f       }
 , { …, …, …, host_square  }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                /* tagged fixnum = 9     */
           ps->r0 = 1+(WORD)&labels[1];    /* tagged return address */
           ps->pc = glo_f;                 /* non-tail call f       */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                    /* add 1 to arg     */
           ps->pc = glo_square;            /* tail call square */
           return;
  }
}
```

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);        /* square arg     */
           ps->pc = ps->r0;                /* return to main */
           return;
  }
}
```

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

1

Execution starts with call to `trampoline` with `ps->pc` referring to program entry point

```
label_struct labels[] = {
   { …, …, …, host_main   } ← ps->pc
 , { …, …, …, host_main   }
 , { …, …, …, host_f      }
 , { …, …, …, host_square }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;               /* tagged fixnum = 9     */
           ps->r0 = 1+(WORD)&labels[1];   /* tagged return address */
           ps->pc = glo_f;                /* non-tail call f       */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                   /* add 1 to arg     */
           ps->pc = glo_square;           /* tail call square */
           return;
  }
}
```

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);       /* square arg     */
           ps->pc = ps->r0;               /* return to main */
           return;
  }
}
```

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

1

```
label_struct labels[] = {
   { …, …, …, host_main   } ← ps->pc
 , { …, …, …, host_main   }
 , { …, …, …, host_f      }
 , { …, …, …, host_square }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];    /* tagged return address */
           ps->pc = glo_f;                 /* non-tail call f        */
           return;

    case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

2

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                    /* add 1 to arg      */
           ps->pc = glo_square;            /* tail call square */
           return;
  }
}
```

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);        /* square arg      */
           ps->pc = ps->r0;                /* return to main */
           return;
  }
}
```

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

1

```
label_struct labels[] = {
   { …, …, …, host_main    }
 , { …, …, …, host_main    }   ← ps->r0
 , { …, …, …, host_f       }   ← ps->pc
 , { …, …, …, host_square  }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                /* tagged fixnum = 9     */
           ps->r0 = 1+(WORD)&labels[1];    /* tagged return address */
           ps->pc = glo_f;                 /* non-tail call f       */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

2

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                    /* add 1 to arg     */
           ps->pc = glo_square;            /* tail call square */
           return;
  }
}
```

3

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);        /* square arg     */
           ps->pc = ps->r0;                /* return to main */
           return;
  }
}
```

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

1

```
label_struct labels[] = {
   { …, …, …, host_main    }
 , { …, …, …, host_main    }   ← ps->r0
 , { …, …, …, host_f       }
 , { …, …, …, host_square  }   ← ps->pc
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];    /* tagged return address */
           ps->pc = glo_f;                 /* non-tail call f        */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

2

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                    /* add 1 to arg      */
           ps->pc = glo_square;            /* tail call square */
           return;
  }
}
```

3

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);        /* square arg      */
           ps->pc = ps->r0;                /* return to main */
           return;
  }
}
```

4

```
void trampoline(processor_state_struct *ps) {
  while (TRUE) GET_HOST(ps->pc)(ps);
}
```

1

```
label_struct labels[] = {
   { …, …, …, host_main   }
 , { …, …, …, host_main   }   <-- ps->pc
 , { …, …, …, host_f      }
 , { …, …, …, host_square }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```

```
void host_main(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {

   case 0: ps->r1 = 9 << 2;                /* tagged fixnum = 9      */
           ps->r0 = 1+(WORD)&labels[1];    /* tagged return address */
           ps->pc = glo_f;                 /* non-tail call f        */
           return;

   case 1: printf("%d\n", ps->r1 >> 2);
           exit(0);
  }
}
```

2

5

```
void host_f(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[2])) / sizeof(label_struct)) {

   case 0: ps->r1 += 4;                    /* add 1 to arg      */
           ps->pc = glo_square;            /* tail call square */
           return;
  }
}
```

3

```
void host_square(processor_state_struct *ps) {
  switch ((ps->pc - (1+(WORD)&labels[3])) / sizeof(label_struct)) {

   case 0: ps->r1 *= (ps->r1 >> 2);        /* square arg      */
           ps->pc = ps->r0;                /* return to main */
           return;
  }
}
```

4

Trampoline Optimizations

1. Locality is exploited by replacing **`return`** with a **`goto`** back to the **`switch`**
2. In *single host* compilation mode, all the code of a compiled file is generated in a single host function (exploits locality of jumps within the file)
3. Computed **`goto`**, available in gcc and clang, is faster than using a **`switch`**
4. Hosts can cache part of the processor state in local variables (improves likelihood C compiler will use machine registers)

Native Backend

- Uses machine registers to store the state of the GVM :

```
rdi    ;; r0
rax    ;; r1
rbx    ;; r2
rdx    ;; r3
rsi    ;; r4
rsp    ;; frame pointer
rbp    ;; heap pointer
rcx    ;; processor state pointer
```

x86-64

- This assignment of machine registers is different than the one used by the C compiler (which we can't and shouldn't control because it would interfere with the optimizations of the C compiler)

Our Approach : Bridges and Dual Purpose `label_struct`

- Use handlers (**bridges**) for cross jumps that move the GVM state where the destination code expects it

- For Native backend : place executable machine code in **`label_struct`**s starting at byte pointed by control point reference (to allow plain *branch-to-address* instructions in native code)

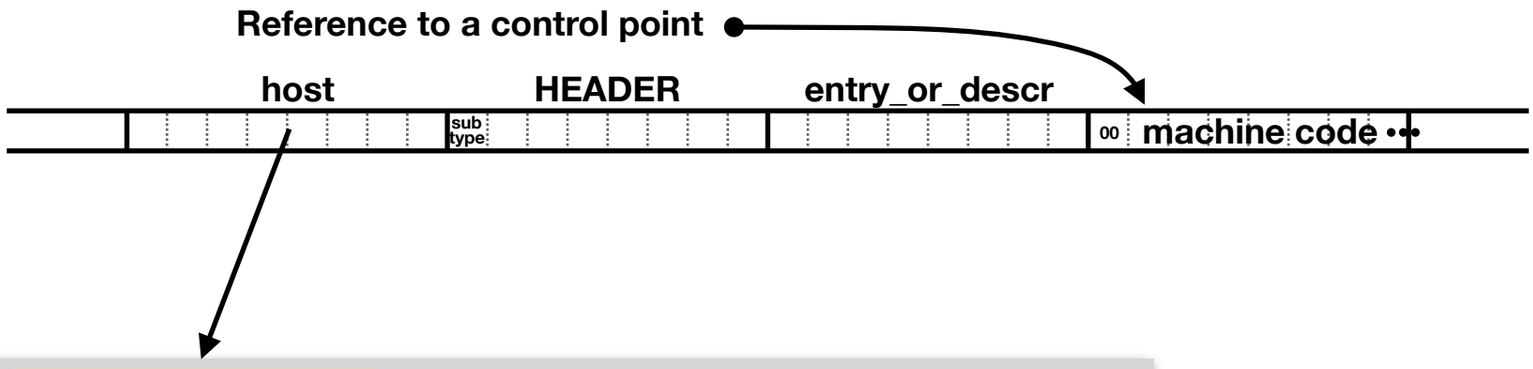
- Both backends use **`label_struct`**s to represent control points :

  - Native backend : host field in **`label_struct`** points to the **`to_native_bridge`**

  - C backend : executable machine code in **`label_struct`** jumps to the **`from_native_bridge`**

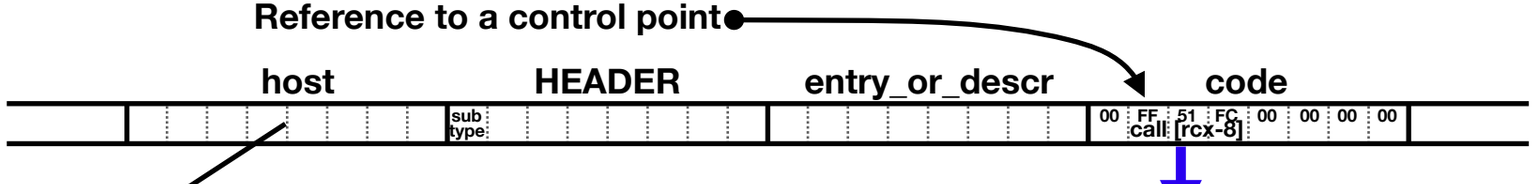
label_struct Layouts

x86-64 backend




```
void to_native_bridge(processor_state_struct *ps) { … }
```

C backend

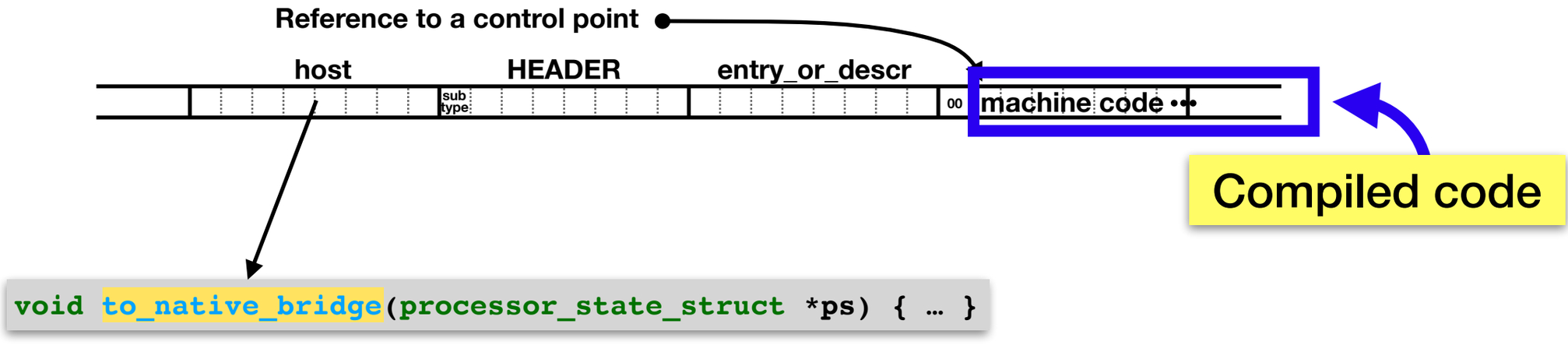



```
void host(processor_state_struct *ps) { … }
```

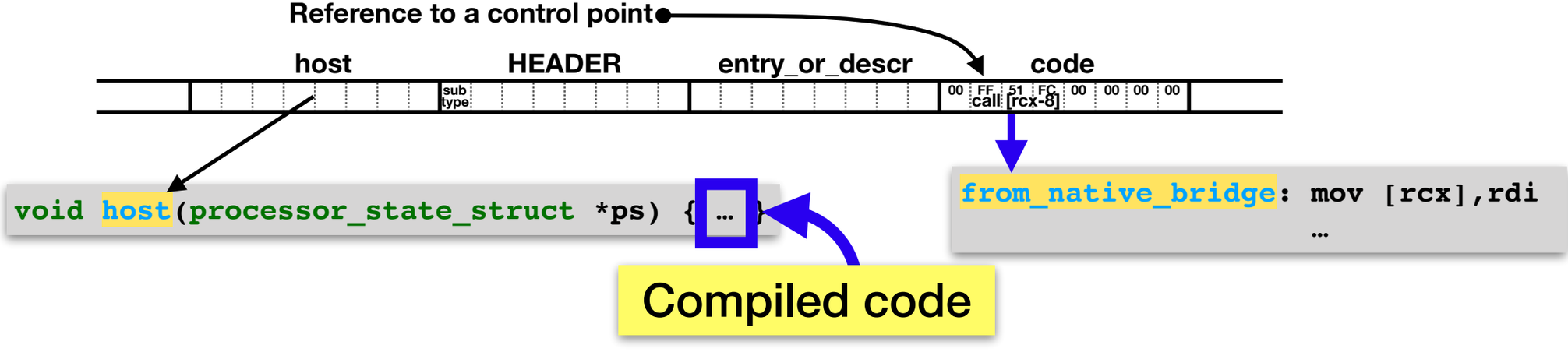
```
from_native_bridge: mov [rcx],rdi
                    …
```

label_struct Layouts

x86-64 backend

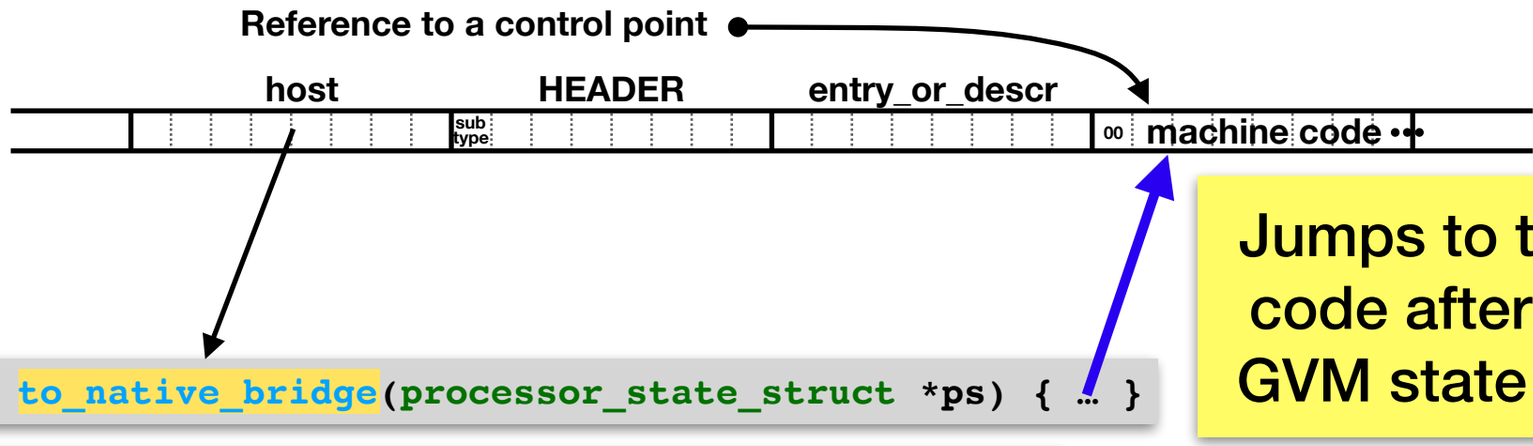


C backend



label_struct Layouts

x86-64 backend

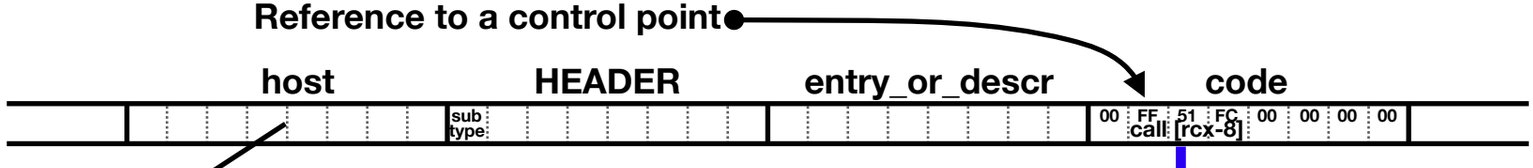



```
void to_native_bridge(processor_state_struct *ps) { … }
```

Jumps to the machine code after storing the GVM state to registers

Bridge called on C to Native cross jump

C backend

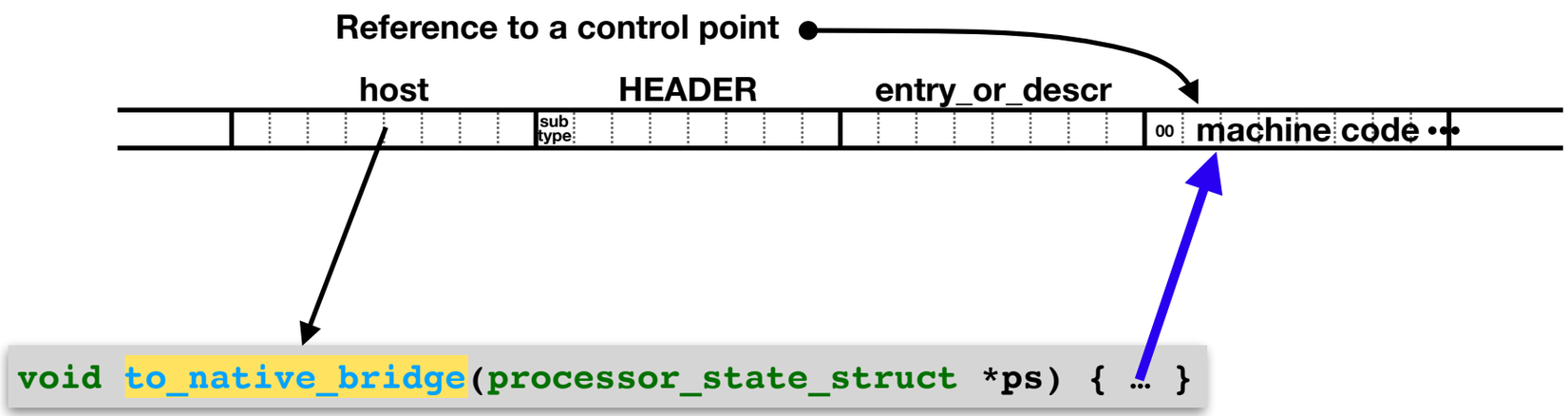



```
void host(processor_state_struct *ps) { … }
```

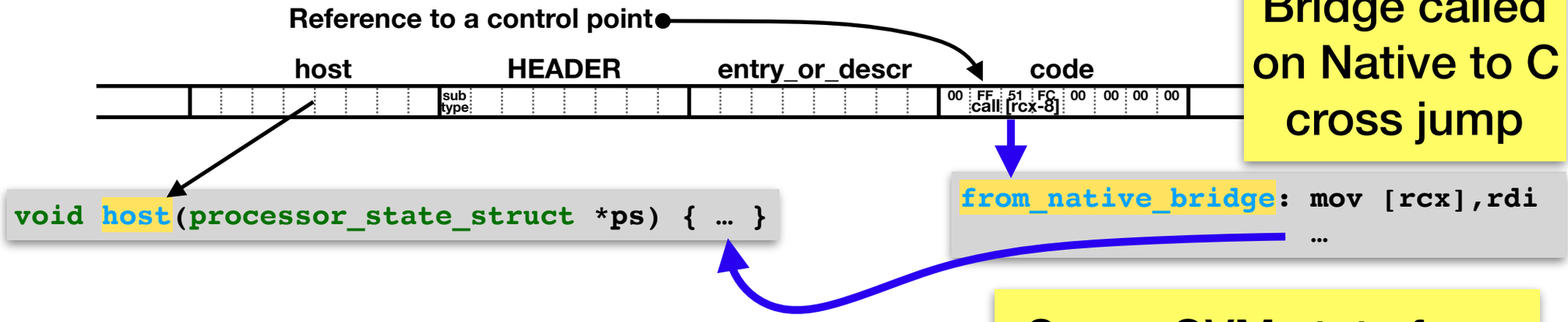
```
from_native_bridge: mov [rcx],rdi
                    …
```

label_struct Layouts

x86-64 backend



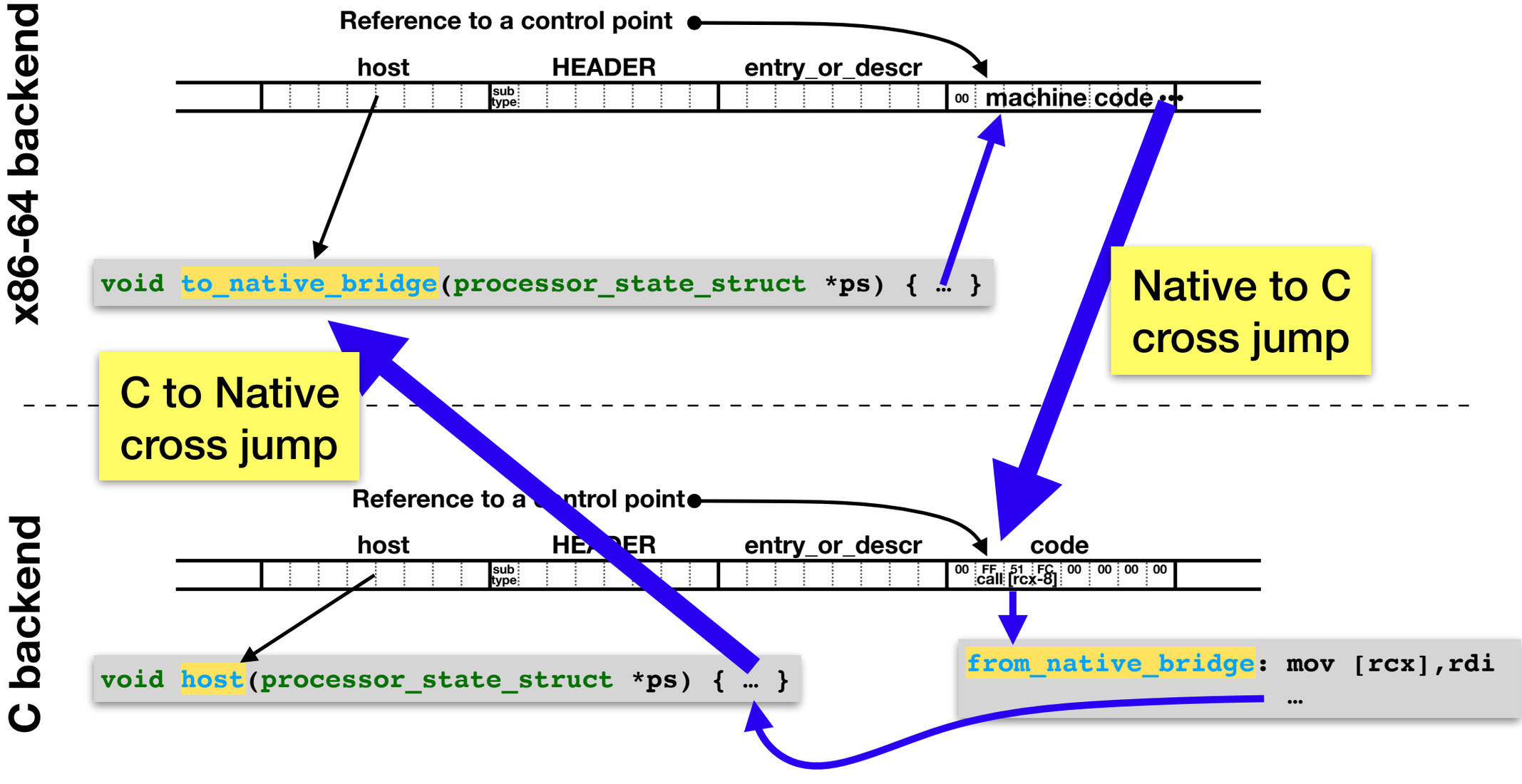
C backend



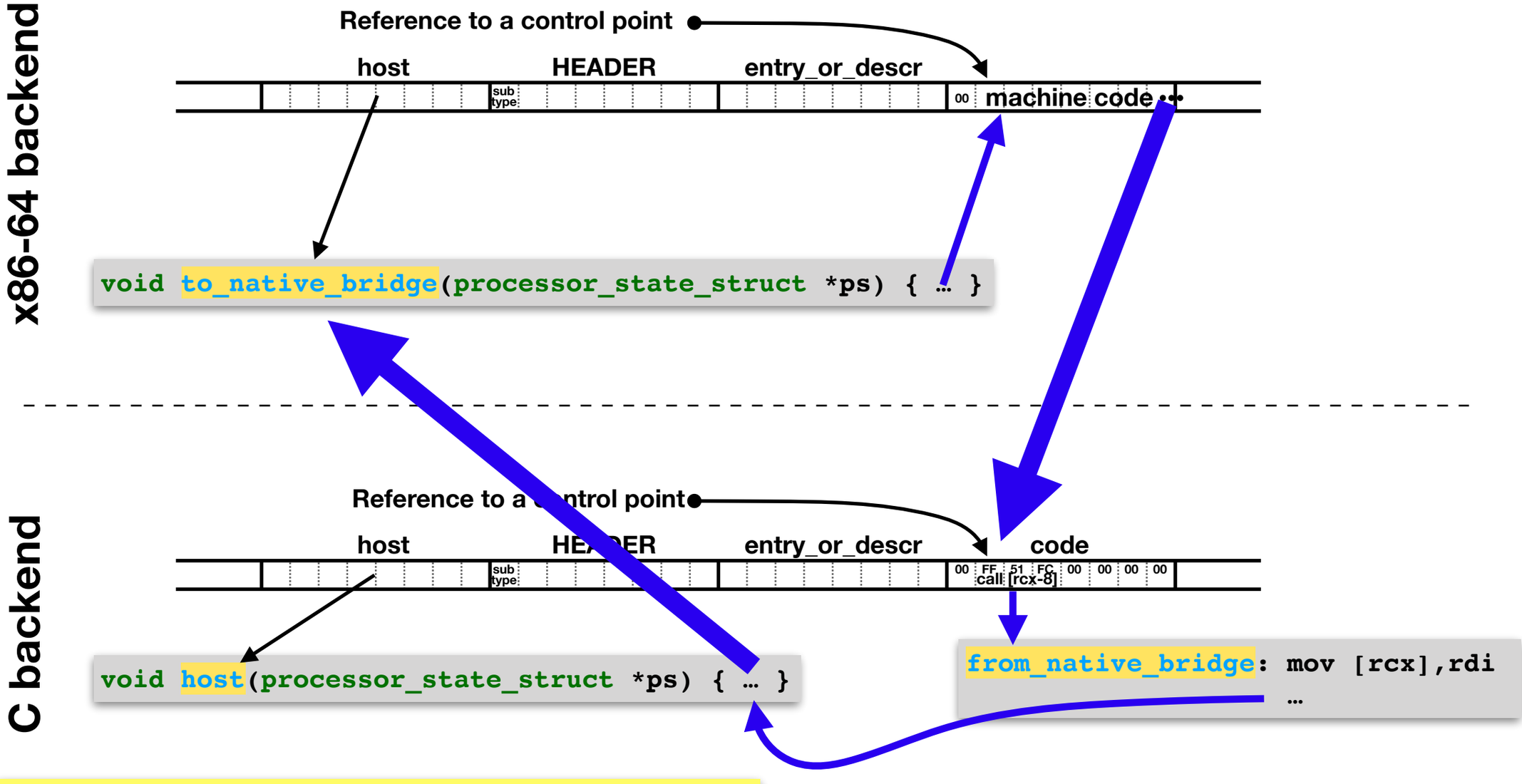
Bridge called on Native to C cross jump

Saves GVM state from registers and jumps to host using trampoline

label_struct Layouts



`label_struct` Layouts



Note: `host` field moved to accommodate closures
`entry` field also needed for closures (see paper)

Implementation of Bridges

x86-64

```
void to_native_bridge(processor_state_struct *ps) {
  __asm__ __volatile__ (

  "mov  %0, %%rcx"  // copy ps into %rcx

  "mov  %%rsp, -2*8(%%rcx)"  // save C sp

  // setup handler for returning from native code
  "lea  from_native_bridge(%%rip), %%rax"
  "mov  %%rax, -1*8(%%rcx)"  // setup handler

  // setup frame pointer and heap pointer registers
  "mov  5*8(%%rcx), %%rsp"  // rsp = ps->fp
  "mov  6*8(%%rcx), %%rbp"  // rbp = ps->hp

  // setup self register
  "mov  4*8(%%rcx), %%rsi"  // rsi = ps->r4

  "mov  8*8(%%rcx), %%rax"  // rax = ps->pc
  "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
  "jl   setup_other_registers"
  "add  $3, %%rsi" // handle closures
  "push %%rsi"
  "add  $-3, %%rsi"

  "setup_other_registers:"
  "mov  (%%rcx), %%rdi"     // rdi = ps->r0
  "mov  1*8(%%rcx), %%rax"  // rax = ps->r1
  "mov  2*8(%%rcx), %%rbx"  // rbx = ps->r2
  "mov  3*8(%%rcx), %%rdx"  // rdx = ps->r3

  "jmp  * 8*8(%%rcx)" // jump to ps->pc


  "from_native_bridge:"

  "mov  %%rdi, (%%rcx)"     // ps->r0 = rdi
  "mov  %%rax, 1*8(%%rcx)"  // ps->r1 = rax
  "mov  %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
  "mov  %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx

  // recover destination control point in ps->pc
  "pop  %%rax"
  "add  $-3, %%rax"  // rax = destination ctrl pt
  "mov  %%rax, 8*8(%%rcx)"   // ps->pc = rax

  "cmpq $0x100000,-1-2*8(%%rax)" // closure?
  "jl   store_self_register"
  "pop  %%rsi"  // handle closures
  "add  $-6, %%rsi"

  "store_self_register:"
  "mov  %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
  "mov  %%rsp, 5*8(%%rcx)"  // ps->fp = rsp
  "mov  %%rbp, 6*8(%%rcx)"  // ps->hp = rbp

  "mov  -2*8(%%rcx), %%rsp"  // restore C sp

  : // no outputs
  : // inputs
    "m" (ps)
  : // clobbers
    "%rdi","%rax","%rbx","%rdx","%rsi",
    "%rcx","%rbp"
  );
}
```

Implementation of Bridges

x86-64

```
void to_native_bridge(processor_state_struct *ps) {
  __asm__ __volatile__ (
  "mov  %0, %%rcx"  // copy ps into %rcx

  "mov  %%rsp, -2*8(%%rcx)"  // save C sp

  // setup handler for returning from native code
  "lea  from_native_bridge(%%rip), %%rax"
  "mov  %%rax, -1*8(%%rcx)"  // setup handler

  // setup frame pointer and heap pointer registers
  "mov  5*8(%%rcx), %%rsp"  // rsp = ps->fp
  "mov  6*8(%%rcx), %%rbp"  // rbp = ps->hp

  // setup self register
  "mov  4*8(%%rcx), %%rsi"  // rsi = ps->r4

  "mov  8*8(%%rcx), %%rax"  // rax = ps->pc
  "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
  "jl   setup_other_registers"
  "add  $3, %%rsi" // handle closures
  "push %%rsi"
  "add  $-3, %%rsi"

  "setup_other_registers:"
  "mov  (%%rcx), %%rdi"     // rdi = ps->r0
  "mov  1*8(%%rcx), %%rax"  // rax = ps->r1
  "mov  2*8(%%rcx), %%rbx"  // rbx = ps->r2
  "mov  3*8(%%rcx), %%rdx"  // rdx = ps->r3

  "jmp  * 8*8(%%rcx)" // jump to ps->pc

  "from_native_bridge:"

  "mov  %%rdi, (%%rcx)"     // ps->r0 = rdi
  "mov  %%rax, 1*8(%%rcx)"  // ps->r1 = rax
  "mov  %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
  "mov  %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx

  // recover destination control point in ps->pc
  "pop  %%rax"
  "add  $-3, %%rax"  // rax = destination ctrl pt
  "mov  %%rax, 8*8(%%rcx)"   // ps->pc = rax

  "cmpq $0x100000,-1-2*8(%%rax)" // closure?
  "jl   store_self_register"
  "pop  %%rsi"  // handle closures
  "add  $-6, %%rsi"

  "store_self_register:"
  "mov  %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
  "mov  %%rsp, 5*8(%%rcx)"  // ps->fp = rsp
  "mov  %%rbp, 6*8(%%rcx)"  // ps->hp = rbp

  "mov  -2*8(%%rcx), %%rsp"  // restore C sp

  : // no outputs
  : // inputs
    "m" (ps)
  : // clobbers
    "%rdi","%rax","%rbx","%rdx","%rsi",
    "%rcx","%rbp"
  );
}
```

to_native_bridge =
18 machine instructions

from_native_bridge =
15 machine instructions

Implementation of Bridges

```
void to_native_bridge(processor_state_struct *ps) {
  __asm__ __volatile__ (
    "mov  %0, %%rcx"  // copy ps into %rcx

    "mov  %%rsp, -2*8(%%rcx)"  // save C sp

    // setup handler for returning from native code
    "lea  from_native_bridge(%%rip), %%rax"
    "mov  %%rax, -1*8(%%rcx)"  // setup handler

    // setup frame pointer and heap pointer registers
    "mov  5*8(%%rcx), %%rsp"  // rsp = ps->fp
    "mov  6*8(%%rcx), %%rbp"  // rbp = ps->hp

    // setup self register
    "mov  4*8(%%rcx), %%rsi"  // rsi = ps->r4

    "mov  8*8(%%rcx), %%rax"  // rax = ps->pc
    "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
    "jl   setup_other_registers"
    "add  $3, %%rsi" // handle closures
    "push %%rsi"
    "add  $-3, %%rsi"

    "setup_other_registers:"
    "mov  (%%rcx), %%rdi"     // rdi = ps->r0
    "mov  1*8(%%rcx), %%rax"  // rax = ps->r1
    "mov  2*8(%%rcx), %%rbx"  // rbx = ps->r2
    "mov  3*8(%%rcx), %%rdx"  // rdx = ps->r3

    "jmp  * 8*8(%%rcx)" // jump to ps->pc

  "from_native_bridge:"

    "mov  %%rdi, (%%rcx)"     // ps->r0 = rdi
    "mov  %%rax, 1*8(%%rcx)"  // ps->r1 = rax
    "mov  %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
    "mov  %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx

    // recover destination control point in ps->pc
    "pop  %%rax"
    "add  $-3, %%rax"  // rax = destination ctrl pt
    "mov  %%rax, 8*8(%%rcx)"   // ps->pc = rax

    "cmpq $0x100000,-1-2*8(%%rax)" // closure?
    "jl   store_self_register"
    "pop  %%rsi"  // handle closures
    "add  $-6, %%rsi"

    "store_self_register:"
    "mov  %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
    "mov  %%rsp, 5*8(%%rcx)"  // ps->fp = rsp
    "mov  %%rbp, 6*8(%%rcx)"  // ps->hp = rbp

    "mov  -2*8(%%rcx), %%rsp"  // restore C sp

    : // no outputs
    : // inputs
      "m" (ps)
    : // clobbers
      "%rdi","%rax","%rbx","%rdx","%rsi",
      "%rcx","%rbp"
    );
}
```

x86-64

`to_native_bridge` =
18 machine instructions

PLUS ONE ITERATION OF THE TRAMPOLINE

`from_native_bridge` =
15 machine instructions

Evaluation

Cost of Jumps

- Compile this Scheme code using C and Native backends and vary where **`dec`** procedure is defined (locally or in runtime system compiled by the C backend)

- 4 possible combinations exercice cross jumps and local jumps (purely in C code, or purely in Native code)

```
(declare (standard-bindings) (not safe))

(define (dec x)   ;; variant of program defines
  (fx- x 1))      ;; dec in the runtime library

(define (run n f)
  (let loop ((n n))
    (if (fx> n 0)
        (loop (f n)))))  ;; call/return dec

(time (run 100000000 dec))
```

Cost of Jumps

cross jumps

	Native to Native	C to C (intra host)	C to C (inter host)	Native to C to Native
x86-64 i7	0.107 **0.07x**	0.238 **0.17x**	1.436 **1x**	2.264 **1.58x**
x86-32 pIII	2.947 **0.17x**	6.787 **0.40x**	16.854 **1x**	19.605 **1.16x**
ARM	2.758 **0.24x**	4.054 **0.35x**	11.431 **1x**	11.680 **1.02x**

Execution time in seconds and time relative to C backend with pure trampoline

Cost of Jumps

cross jumps

	Native to Native	C to C (intra host)	C to C (inter host)	Native to C to Native
x86-64 i7	0.107 **0.07x**	0.238 **0.17x**	1.436 **1x**	2.264 **1.58x**
x86-32 pIII	2.947 **0.17x**	6.787 **0.40x**	16.854 **1x**	19.605 **1.16x**
ARM	2.758 **0.24x**	4.054 **0.35x**	11.431 **1x**	11.680 **1.02x**

Execution time in seconds and time relative to C backend with pure trampoline

- Cross jumps cost **2%** to **58%** more than pure trampoline

Cost of Jumps

cross jumps

	Native to Native	C to C (intra host)	C to C (inter host)	Native to C to Native
x86-64 i7	0.107 **0.07x**	0.238 **0.17x**	1.436 **1x**	2.264 **1.58x**
x86-32 pIII	2.947 **0.17x**	6.787 **0.40x**	16.854 **1x**	19.605 **1.16x**
ARM	2.758 **0.24x**	4.054 **0.35x**	11.431 **1x**	11.680 **1.02x**

Execution time in seconds and time relative to C backend with pure trampoline

- Local jumps within code generated by Native backend are up to **14x** faster than inter host trampoline and **2.4x** faster than intra host trampoline

Conclusion

Desiderata

1. **Cross jumps are transparent** (All control points support both the trampoline and *branch-to-address*)

2. **Native code jumps are plain *branch-to-address* instructions** (The single machine jump instruction is optimal)

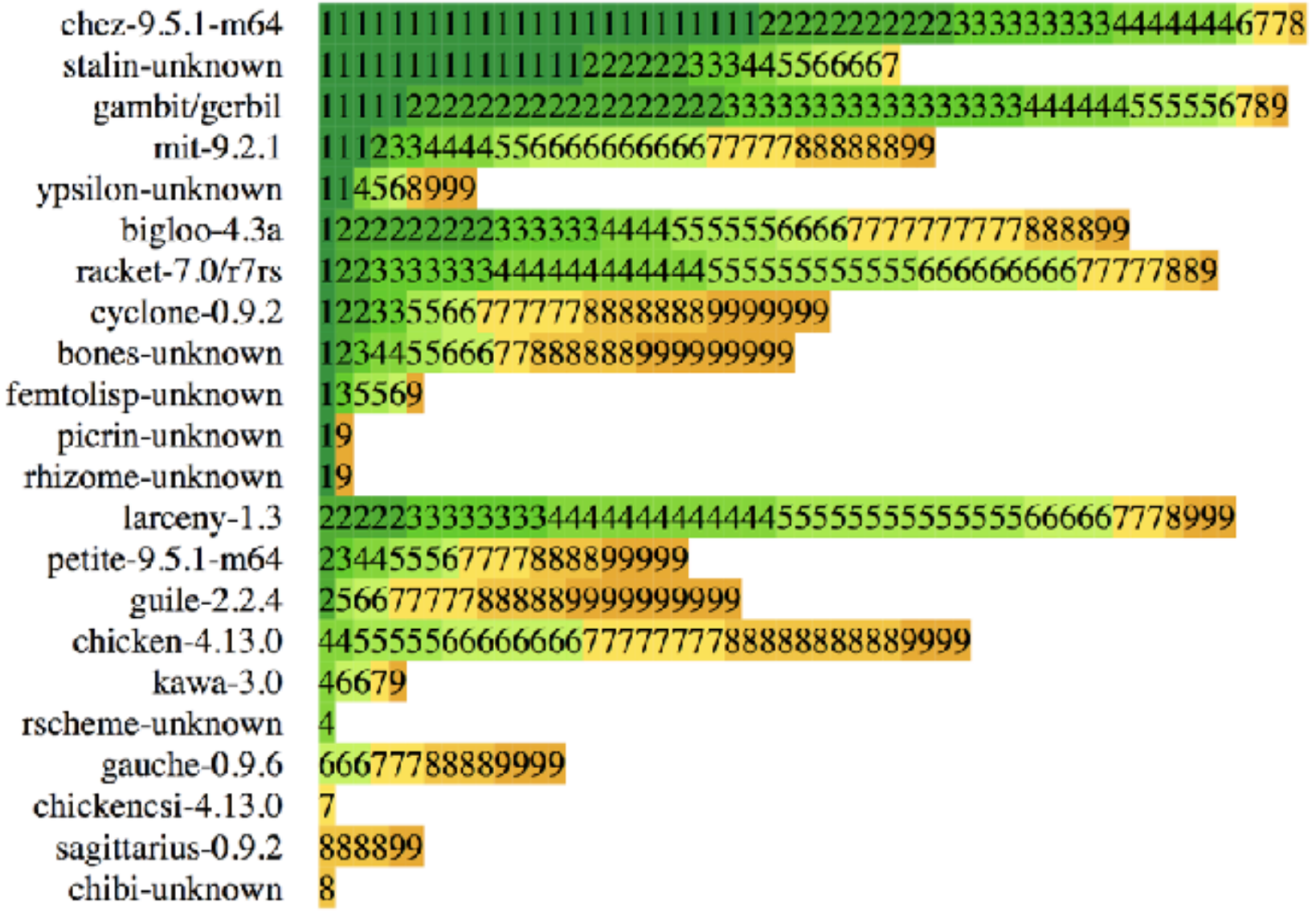
3. **Cost of cross jumps is similar to C backend local jumps** (Only 1.58x slower for x86-64, 1.16x for x86-32 and 1.02x for ARM)

Use cases

1. **Useful tool for native backend development (all system features available throughout development)**

2. **Possibly completely avoid reimplementing the I/O and other complex subsystems**

Motivation for Native Backend



Motivation for Native Backend

	C Backend	x86-64 Backend	Chez Scheme 9.5.1 --optimize-level 3
`ack`	10.943 **1.13x**	9.696 **1x**	12.392 **1.28x**
`fib`	2.989 **1.19x**	2.507 **1x**	4.249 **1.69x**
`tak`	7.628 **0.98x**	7.746 **1x**	8.135 **1.05x**
`takl`	3.993 **1.02x**	3.906 **1x**	7.576 **1.94x**
`sum`	2.878 **0.77x**	3.755 **1x**	11.784 **3.14x**

mostly loops

Unsafe fixnum benchmarks with frequent jumps
Gambit v4.9.0

(Preliminary results)