Optimizing Scheme, part I
cons should not cons its arguments, part I
a Lazy Alloc is a Smart Alloc

Alex Gal
COMP 621
cancelled

Optimizing Scheme, part II
an inexistant return is a smart return

Samuel Gélineau
COMP 621
February 7, 2008

stack-storage optimization for short-lived data
a one slide summary

- most object are short-lived
- allocate them on the stack (faster than malloc)
- those that outlive the function call are moved to the heap
- that's quite a short zeroth generation!

cons should not cons its arguments, part II
Chenev on the M.T.A.

Henry Baker

Sing along!

Charlie on the M.T.A.
oh, will he ever return?
no, he’ll never return.
and his fate is still unlearned,
he’s a man who’ll never return!
Compiling Scheme to C
Scheme and C are so different

Scheme
High-level, recursive, lots of small garbage-collected conses.

```scheme
(define (reverse a-list)
  (append (reverse (cdr a-list))
          (list (car a-list))))
```

C
Hand-optimized low-level details.

```c
void reverse(int* array, int length) {
    for(int i = 0, i = length-1; i < i; ++i, --i) {
        swap(&array[i], &array[i+1]);
    }
}
```

No way our generated code can pull that sort of trick!

Features only provided by Scheme
apart from allowing weird characters in identifiers

continuations
```
(define labels (make-hash-table))

(define (label name)
  (call/cc (lambda (cc)
             (hash-table-put! labels name cc)
             (cc 'label-return-value))))

(define (goto name)
  (let ((cc (hash-table-get labels name)))
      (cc 'label-return-value))
```

Features only provided by C
apart from segfaults

```c
#include

int handler_depth = 0;

int try(void (*body)(void)) {
    int error_code = setjmp(handlers[+handler_depth]);
    if (error_code == EXIT_SUCCESS)
        body();
    return error_code;
}

do throw(int error_code) {
    if (error_code != EXIT_SUCCESS)
        longjmp(handlers[handler_depth--], error_code);
}
```

a Scheme-specific optimization
required by the language definition, but not always strictly obeyed

C
```c
void recursive_loop() {
    printf("infinite bottles of beer on the wall\n")
    recursive_loop();  // exhausts the stack
}
```

Scheme
```scheme
(define (recursive-loop)
  (display "infinite bottles of beer on the wall\n")
  (recursive-loop)) : does not exhaust the stack!

(recursive-loop)
```

tail-call optimization
a C-specific optimization
not standard, but implemented by most compilers

C
{
    int n;
    int *a = &n;        *a = 42;
    int *b = malloc(sizeof(int));    *b = 43;
    int *c = calloc(sizeof(int));    *c = 44;
    printf("%d %d %d\n", *a, *b, *c);
}
*a and *c are freed at the end of the block, but not *b.

Scheme
Garbage-collection: when all you have is a hammer...

Target code for tail-recursion
a bit of interpreter overhead in the compiled code

trampoline
void* args;
void* result;
typedef void* (*bounce)();

void* recursive_loop()
    printf("infinite bottles of beer on the wall\n");
    return recursive_loop;
}

void trampoline()
    bounce f = recursive_loop;
    for(;;)
        f = f();

Amortizing the trampoline cost
"avoid making a large number of small trampoline bounces
by occasionally jumping off the Empire State Building"

Bungee
membuf trampoline:

void recursive_loop() {
    int n;
    printf("infinite bottles of beer on the wall\n");
    if (n > STACKLIMIT)
        longjmp(trampoline, (int) recursive_loop);
    else
        recursive_loop();
}

int main() {
    bounce f = (bounce) setjmp(trampoline);
    if (f == NULL) f = &recursive_loop;
    f();
}

Garbage-collecting the stack
don’t throw the live variables with the bathwater

a longer zeroth generation
if (n > STACKLIMIT) {
    *c();
    alloc(-STACK_SIZE);
} recursive_loop():

Move live variables to the heap, garbage-collect the rest.
Using a copy-collector, young dead nodes are collected for free!
Continuation-passing-style
What if the entire program was written by a tail-call fanatic?

let all calls be tail calls
(define (if cond cc then cc else cc cc)
  (cond cc (lambda (bool)
          (if bool
               (then cc cc)
               (else cc cc))))

(define (+ rand1 cc rand2 cc cc)
  (rand1 cc (lambda (n1)
        (rand2 cc (lambda (n2)
                 (cc (+ n1 n2)))))))

Bungeeesh!
a one slide summary

- never return. never.
- use continuation-passing-style to avoid returns.
- always allocate on the stack.
- when we run out of stack space:
  - flush the dead nodes (for free)
  - copy the live nodes (amortized by the mallocs we avoided)
  - flush the call stack (dec RESP KEEP STACK_SIZE)
  - call the continuation