Optimizing Scheme, part I

`cons` *should not* `cons` *its arguments, part I*

a Lazy Alloc is a Smart Alloc

Alex Gal

COMP 621

cancelled

- 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!

Optimizing Scheme, part II

an inexistant `return` *is a smart* `return`

Samuel Gélineau

COMP 621

February 7, 2008

`cons` *should not* `cons` *its arguments, part II*

Cheney on the M.T.A.

Henry Baker

ACM *Sigplan Notices* 30(9), 1995

cons *should not* cons *its arguments, part II*

Cheney 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!

Scheme

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

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

C

Hand-optimized low-level details.

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

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

longjmp

```
jmp_buf handlers[MAX_DEPTH];
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;
}

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

Features only provided by C
apart from segfaults

longjmp

```c
jmp_buf handlers[MAX_DEPTH];
int handler_depth = 0;

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

Question to the audience

C-only?

Isn't this equivalent to an escape continuation?

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

longjmp

```
jmp_buf handlers[MAX_DEPTH];
int handler_depth = 0;

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

Question to the audience

C-only?

Isn't this equivalent to an escape continuation?

Almost, but the abstraction level is different.

```
void 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

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

Scheme

```
(define (recursive-loop)
  (recursive-loop)   ; exhausts the stack
  (display "infinite bottles of beer on the wall\n"))

(recursive-loop)
```

C

```
void recursive_loop() {

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

tail-call optimization

Scheme

```
(define (recursive-loop)

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

C

```
void recursive_loop() {

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

Question to the audience

Language definitions usually specify semantics, not optimizations.

What pushed the language designers to do this?

```
he wall\n")
ack!
(recursive-loop)
```

a Scheme-specific optimization

required by the language definition, but not always strictly obeyed

C

```
void recursive_loop() {

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

Question to the audience

Language definitions usually specify semantics, not optimizations.

What pushed the language designers to do this?

Lack of iteration. If recursion is to take on the role of `for`-loops, they better be efficient.

```
he wall\n")
ck!
(recursive-loop)
```

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 = alloca(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

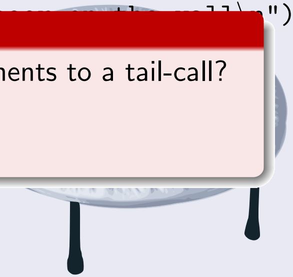
a bit of interpreter overhead in the compiled code

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();
}
```

trampoline

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

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



Question to the audience

Can local variables be passed as arguments to a tail-call?

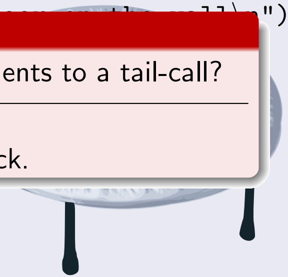
```
  bounce f = recursive_loop;
  for(;;)
    f = f();
}
```

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");
```



Question to the audience

Can local variables be passed as arguments to a tail-call?

With pass-by-value *only*.

conses cannot be allocated on the stack.

```
  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

```
jmp_buf trampoline;

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

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

Amortizing the trampoline cost

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

bungee

```
jmp_buf trampoline;

void recursive_loop() {
  int _;
  printf("infinite bottles of beer on the wall\n");
  if (&_ > STACK_LIMIT)
```

Question to the audience

Now, can local variables be passed by reference?

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

Amortizing the trampoline cost

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

bungee

```
jmp_buf trampoline;

void recursive_loop() {
  int _;
  printf("infinite bottles of beer on the wall\n");
  if (&_ > STACK_LIMIT)
```

Question to the audience

Now, can local variables be passed by reference?

No, since the bungee jump will unpredictably free them. Still no `alloca` optimization in sight.

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

a longer zeroth generation

```
if (&_ > STACK_LIMIT) {
  gc();
  alloca(-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!

Garbage-collecting the stack

don't throw the live variables with the bathwater

a longer zeroth generation

```
if (&_ > STACK_LIMIT) {
  gc();
  alloca(-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!

Question to the audience

Now, can local variables be passed by reference?

a longer zeroth generation

```
if (&_ > STACK_LIMIT) {
  gc();
  alloca(-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!

Question to the audience

Now, can local variables be passed by reference?

No, since not all calls are tail-calls!

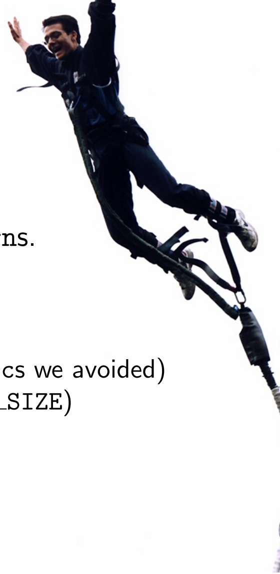
Continuation-passing-style

What if the entire program was written by a tail-call fanatic?

let all calls be tail calls

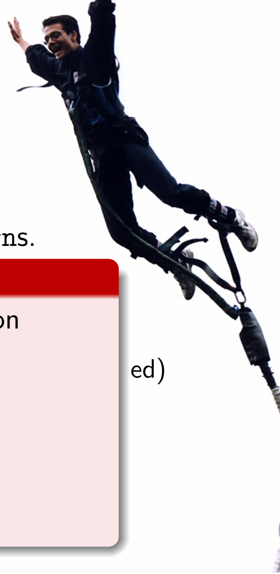
```
(define (_if cond_cc then_cc else_cc cc)
  (cond_cc (lambda (bool)
             (if bool
               (then_cc cc)
               (else_cc cc)))))

(define (_+ rand1_cc rand2_cc cc)
  (rand1_cc (lambda (n1)
              (rand2_cc (lambda (n2)
                          (cc (+ n1 n2)))))))
```

Bungeeeeee!
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 %ESP %ESP STACK_SIZE`)
  - call the continuation

Bungeeeeee!
a one slide summary



- never return. *never*.
- use continuation-passing-style to avoid `returns`.

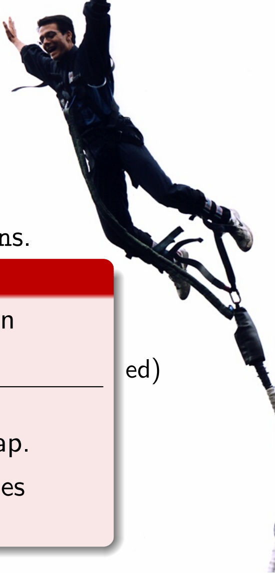
Question to the audience

What is the difference between part I's optimization and part II's?

ed)

Bungeeeeee!

a one slide summary



- never return. *never*.
- use continuation-passing-style to avoid `returns`.

Question to the audience

What is the difference between part I's optimization and part II's?

Part I's allowed baby nodes to die on the stack, but longer-lived nodes had to be evicted to the heap.

Part II's continuation-passing-style allows teen nodes to die on the stack too. Hurray!

ed)