The Layers of Larceny's FFI

Felix S Klock II
pnkfelix@ccs.neu.edu
Northeastern University

Larceny's Foreign Function Interface

- Larceny
  - GC, compiler research
  - FFI *cannot* constrain system design
- Experience report
  - For implementors
  - . . . and curious users

Experience report on design and implementation of Larceny's FFI.
(just describing design choices; I'm pretty certain none are innovations)
(What parts worked well for us)

Foreign Function Interfaces: Why

- Low-level facilities (. . . but but but!)
- Code reuse!

Access to low-level facilities via interfaces targeting other languages (e.g. C)

Do not reimplement OpenGL, GTK+, etc
Life is too short!

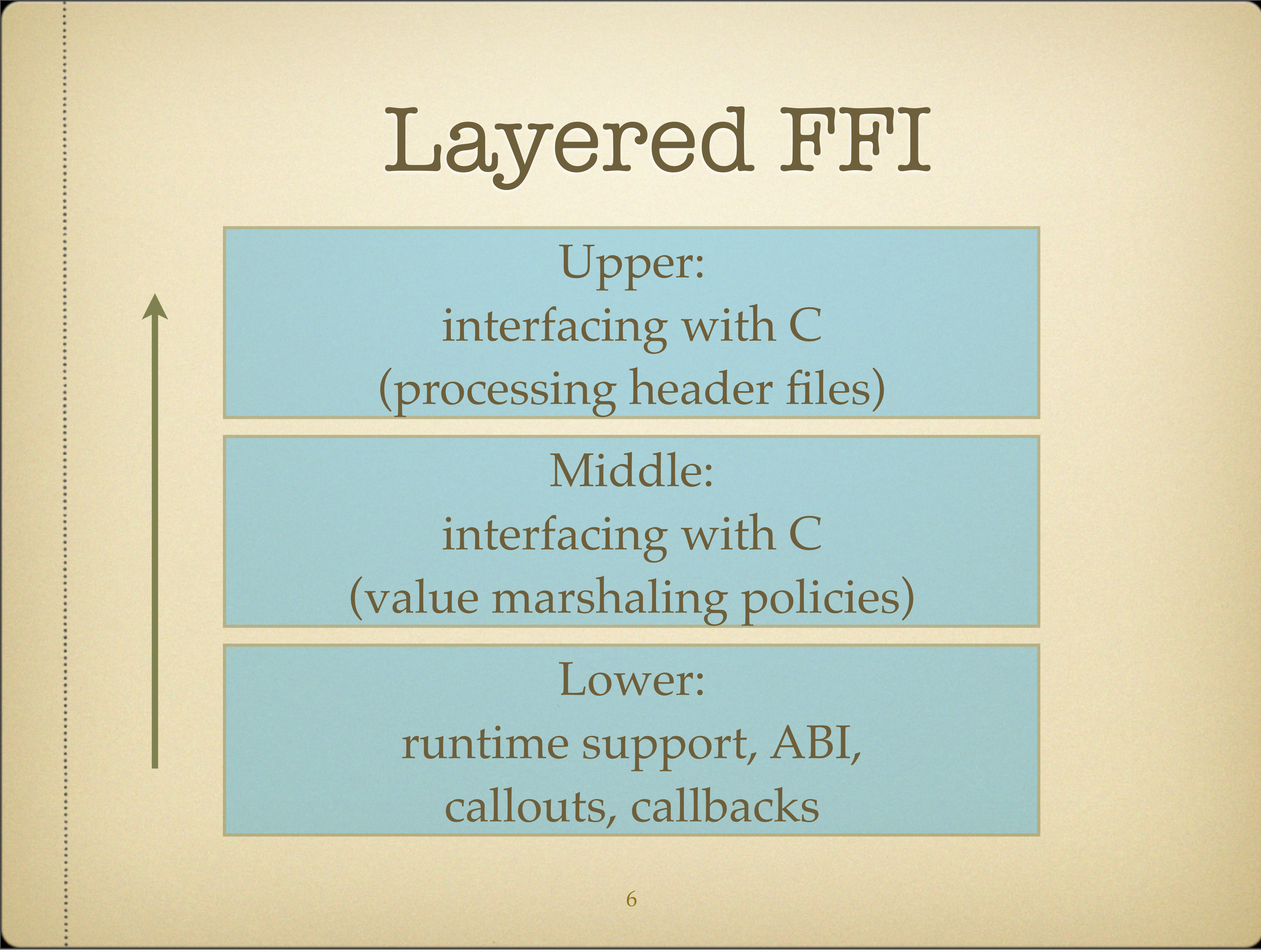
Goals for Larceny FFI

- Constraint: precise, copying garbage collector
- FFI design *cannot* constrain Larceny VM design
- Scheme closures as C function pointers ("callbacks") as well as "callouts"
- Write glue in Scheme, not C

Side-benefits of Larceny FFI

- Automatic relinking on heap reload
- Header file processing
- Support code oblivious to Larceny VM design

Latter is not a user-visible benefit; it is solely appreciated by the Larceny developers.



Presentation won’t address the middle (it is discussed in the paper)

Layered FFI

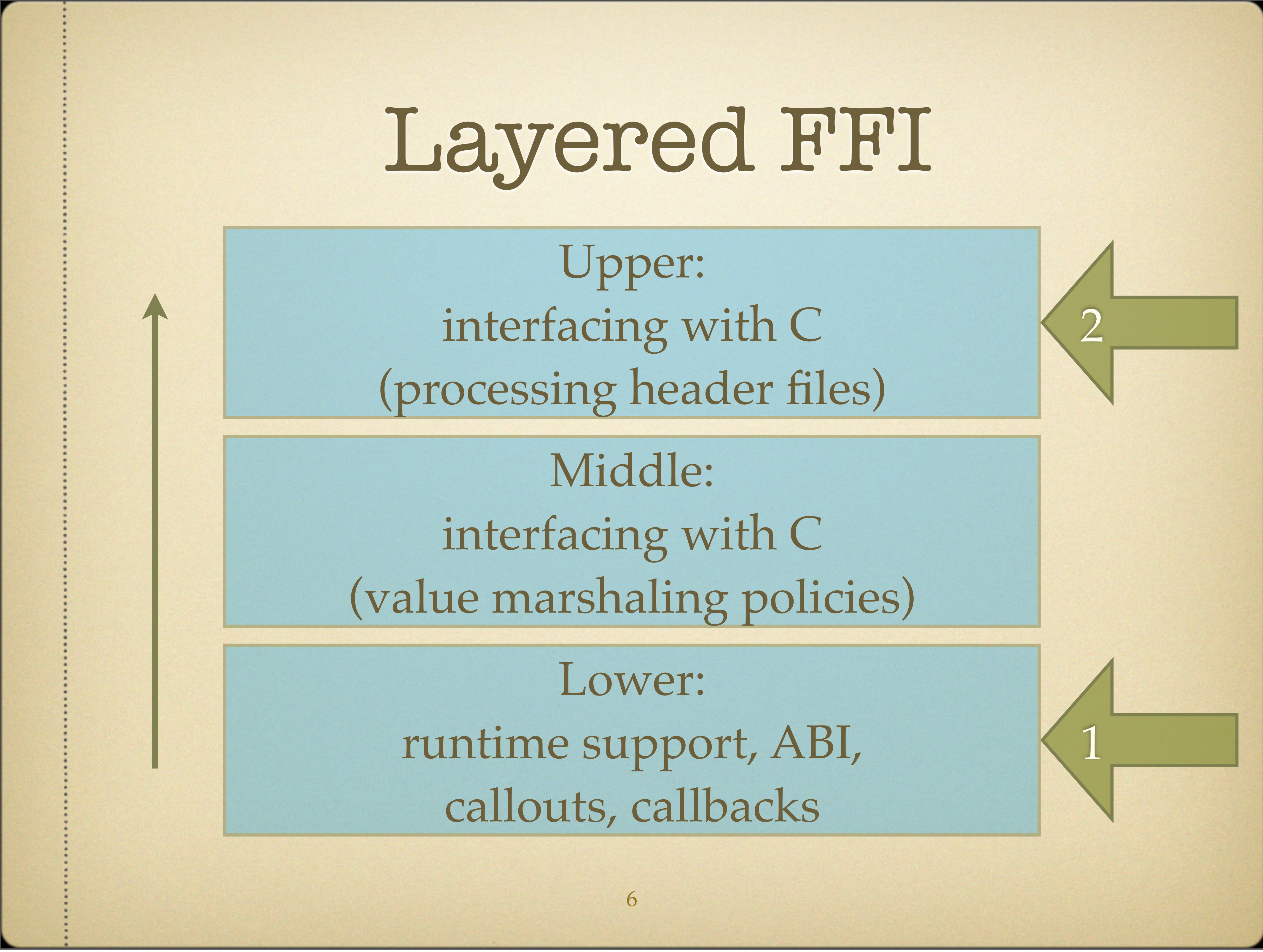
Upper:
interfacing with C
(processing header files)

Middle:
interfacing with C
(value marshaling policies)

Lower:
runtime support, ABI,
callouts, callbacks

1

Presentation won’t address the middle (it is discussed in the paper)



Presentation won’t address the middle (it is discussed in the paper)

Larceny Architecture

Larceny Architecture

- Larceny
  - Virtual Machine (aka VM)
  - Runtime (supports VM); written in mostly C
- Register assignment
- Calling convention

Virtual Machine has own processor configuration for running compiled Scheme code.
Larceny Runtime is mostly implemented in C and thus adheres to ABI specifications.
Register usage conventions. (E.g. Larceny chooses all reg roles; C: registers ABI specified.) E.g. prev mapped %sp to GLOBALS array
Calling conventions: all parameters are caller-save in Larceny.
Scheme code evaluated in MacScheme, but Runtime (FileSystem interaction & GC) are part of the C world.
Consider a change-directory operation...

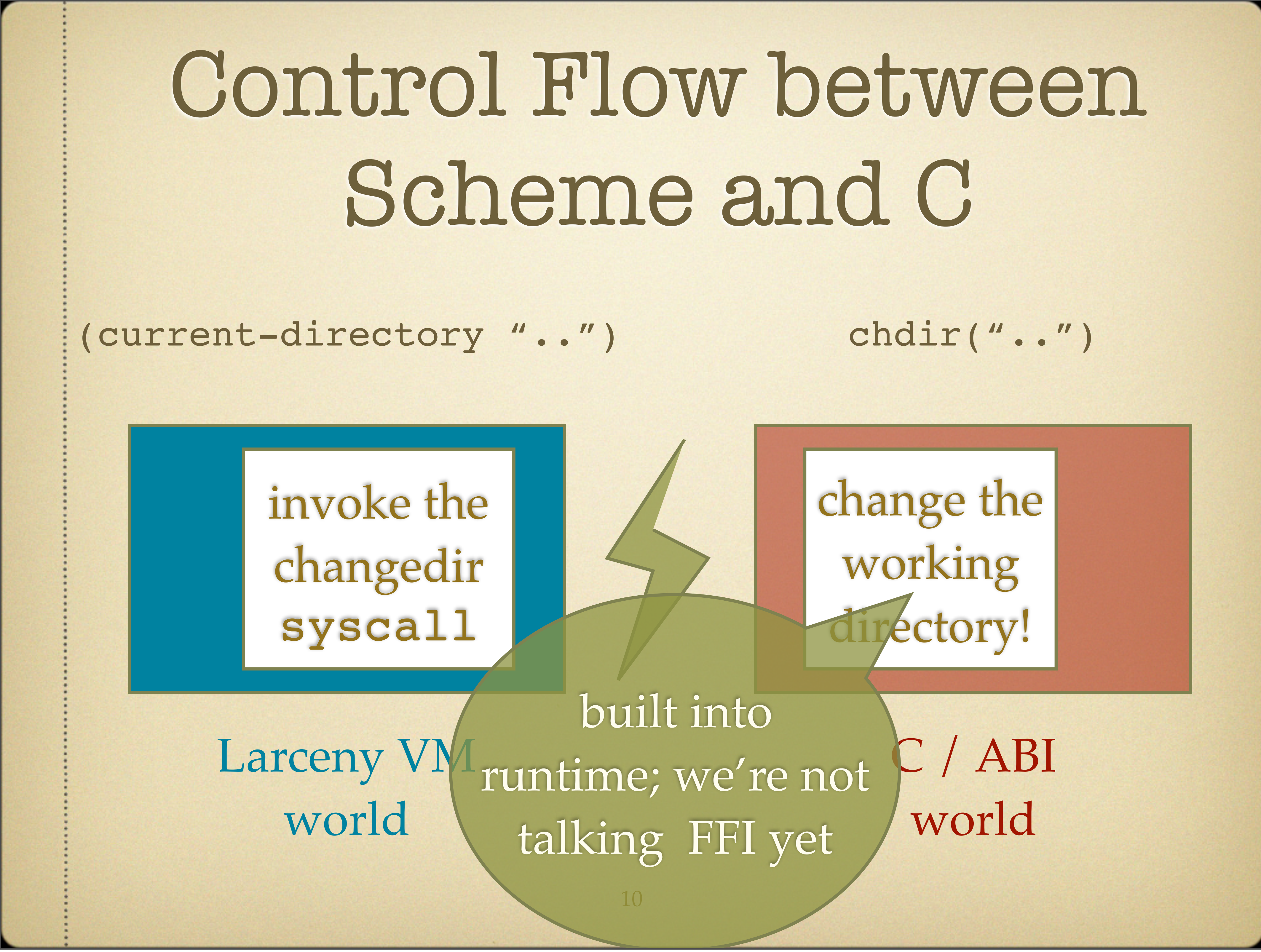
Context Switches

- Some functionality outside compiled Scheme
  - File system commands
  - Garbage collector interactions
- Implemented by Larceny runtime
- Larceny Scheme `syscall` procedure

Control Flow between Scheme and C

`(current-directory "..")`   `chdir("..")`

Low-level operation like "chdir": shift the processor state so runtime C code happy with invocation context. Likewise, return to Scheme must shift processor back to MacScheme-compatible state. All *already* implemented in Larceny's syscall support.



Low-level operation like "chdir": shift the processor state so runtime C code happy with invocation context. Likewise, return to Scheme must shift processor back to MacScheme-compatible state. All *already* implemented in Larceny's syscall support.

Low Level Challenges

Our tasks

- Callout: given C function (name/address) *and* its signature, create compatible Scheme procedure
- Callback: given Scheme closure *and* C function signature, create C function pointer that invokes the closure

A use case

```
(define-gtk-enum gtkwindowtype
  (toplevel "GTK_WINDOW_TOPLEVEL")
  (popup    "GTK_WINDOW_POPUP"))

(define gtk-window-new
  (foreign-procedure “gtk_window_new”
                     ‘(gtkwindowtype) gtkwindow*))
```

library code

---

client code

```
(define window (gtk-window-new ‘toplevel))

(define (key-press w e)
  (write `(key-press ,(gdk-event-keyval e)))
  (newline))

(g-signal-connect window
                  “key_press_event” key-press)
```

Why this is hard

- Value correspondence (“Symbol? Pair?”)
- Value formats differ (fixnum bitwidth, tags)
- VM mismatch
- No `apply` in C
- C function pointers are *only* code; Scheme closures are code plus environment

Scheme values are tagged; C’s are not.
VM invocation is not a C invocation; and must establish proper processor context
C does not have a way to apply function to a *package* holding its arguments

Some solutions. . .

Problem	Solution
Value domains and formats differ	Map to/from primitive domains, strip/add tags
VM mismatch	Reuse runtime context switch from syscalls
No `apply` in C	?
C function pointers are not Scheme closures	?

C does not have apply ...

- C alone can only approximate `apply` (poorly) via fixed size dispatch table
- Plus, types matter
  - `float` not same as `int`
  - One `long long` not same as two `long`'s
- 4 types, 10 args : 1,048,576 entries

16

… so make an apply elsewhere

- Given address & signature of foreign function *f*
- Construct machine code for "C function" *g*
  - *g* takes array holding arguments to *f*
  - *g* places arguments according to ABI calling convention, and then invokes *f*
- (more like specialized `apply`$_f$ than `apply`)

...and one more thing

- Callout/callback glue generation is implemented *as Scheme code*
- Machine code held in heap-allocated bytevectors!
  - garbage collectable
  - (but nonrelocatable)
- Do not be fooled: *g* expects to run in C context

See paper for details.

...and one more thing

- Callout / callback glue generation is

```
list->bytevector
'(#x55                     ; PUSH EBP                standard prologue
  #x89 #xE5                ; MOV EBP, ESP
  ,@(make-filler tr #x55)  ; PUSH EBP (filler to 16-byte aligned)
  #x53                     ; PUSH EBX
  #x56                     ; PUSH ESI
  #x8B #x75 #x08           ; MOV ESI, [EBP+8]        load argv
  #x81 #xEC ,@args-size    ; SUB ESP, 4*argc         allocate space for args
  #x8B #xFC                ; MOV EDI, ESP            copy pointer
  #xFC)))                  ; CLD                     copy upward
```

  - (but nonrelocatable)
- Do not be fooled: *g* expects to run in C context

See paper for details.

...and one more thing

- Callout/callback glue generation is implemented *as Scheme code*
- Machine code held in heap-allocated bytevectors!
  - garbage collectable
  - (but nonrelocatable)
- Do not be fooled: *g* expects to run in C context

See paper for details.

(alternatives, but ...)

- `apply` not expressible in C, but g = $\texttt{apply}_f$
- *Could* generate C code for *g*, compile, and dynamically link into running Larceny system
- But that requires users to have C compiler available
- Plus: dynamic code generation solves callback problem (encode closure address in *g*-code)

Problems, Solutions

Problem	Solution
Value domains and formats differ	Map to/from primitive domains, strip/add tags
VM mismatch	Reuse runtime context switch from syscalls
No `apply` in C	Generate callout *g*-code from signature
C function pointers are not Scheme closures	Generate callback *g*-code from signature

Scheme values are tagged; C's are not.

Callout Creation, Usage

Constructs glue (as in *g*-code) for C opendir function

```
> (define unix/opendir
    (foreign-procedure "opendir" '(string) 'uint))
#<PROCEDURE>
```

Invocation of `unix/opendir` passes *g* and marshaled arg list to Runtime syscall ffi callout; returns `dir_ent*` (aka “`uint`”)

```
> (unix/opendir “/tmp”)
1050560
```

Callout Control Flow

(for foreign function *f*)

Callout Control Flow

(for foreign function *f*)

Callout/Callback glue follows C ABI

- Ten years ago there was no x86 *native* Larceny VM (only C backend)
- We recently implemented x86 native
- FFI continued working

Tell story of:
1. Lars dev'd x86 FFI atop Petit Larceny
2. Felix dev'd native x86 Larceny
3. Lars' x86 FFI worked transparently, orthogonal to transition!

Why again?

- Heap dump/reload (with library reloading); see paper
- Glue uses ABI call convention, *not* Larceny VM call convention (robust to VM design changes)
- FFI does not constrain Larceny VM design
- Drawback: generating machine code for *g* ourselves (instead of using e.g. `libffi`)

We've adopted this control/heap structure for a number of reasons; the tramp objects allow us to relink foreign objects during a heap load. (See paper for more details.)
I cannot stress enough the idea of separating the MacScheme calling convention from the C calling convention. (It took me a long time to understand, and longer to appeciate.)
On the drawback: there is significant initial development (and also maintenance) overhead for our FFI design. E.g. we still do not have a PowerPC port of the FFI.

High Level Interfacing Problems

Now that I've shown you the low level details of the kernel of the Larceny FFI, lets talk about a higher level problem and the solution we adopted.

How to hack with C?

- Many libraries have implementation (shared object code) and interface (C header file)
- Would be nice if interface were not encoded as a C header file
  - (see FFIGEN Manifesto [Hansen,1996])
- We accept standard operating procedure and treat the header file as the expected interface

Two kinds of libraries

- C libraries with *all* functionality exported via functions over “simple” types
- C libraries assuming that clients can/will use constants and type definitions of *header files*

27

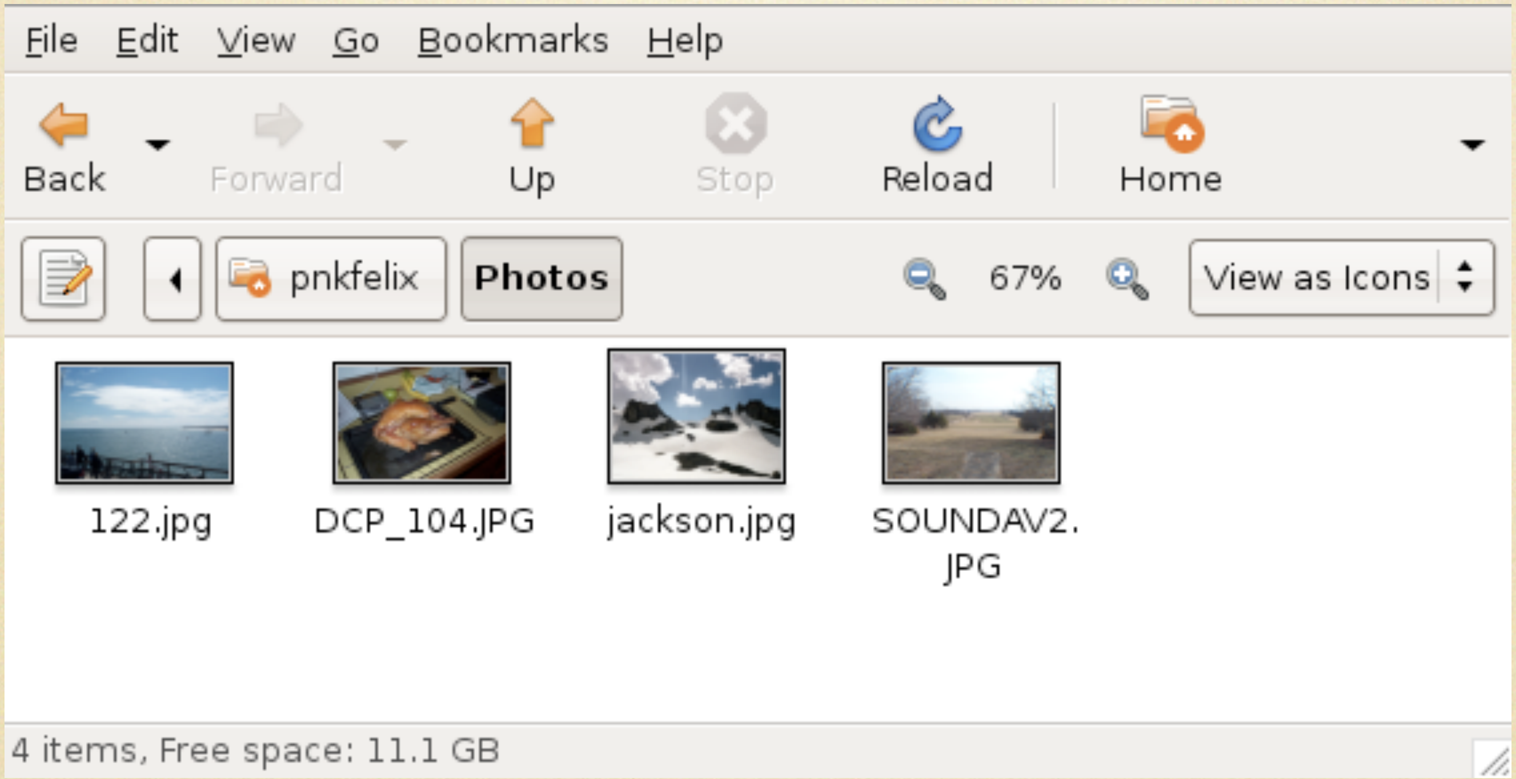
Very simple FFI’s suffice for the former category.
Unfortunately, vast majority of libraries fall into latter category

Example: The UNIX Filesystem

- A tale from the Larceny source tree
- Larceny does not have a `list-directory` primitive operation or syscall
- But FFI-based implementation was available

Felix decided to see if the code for list-directory worked

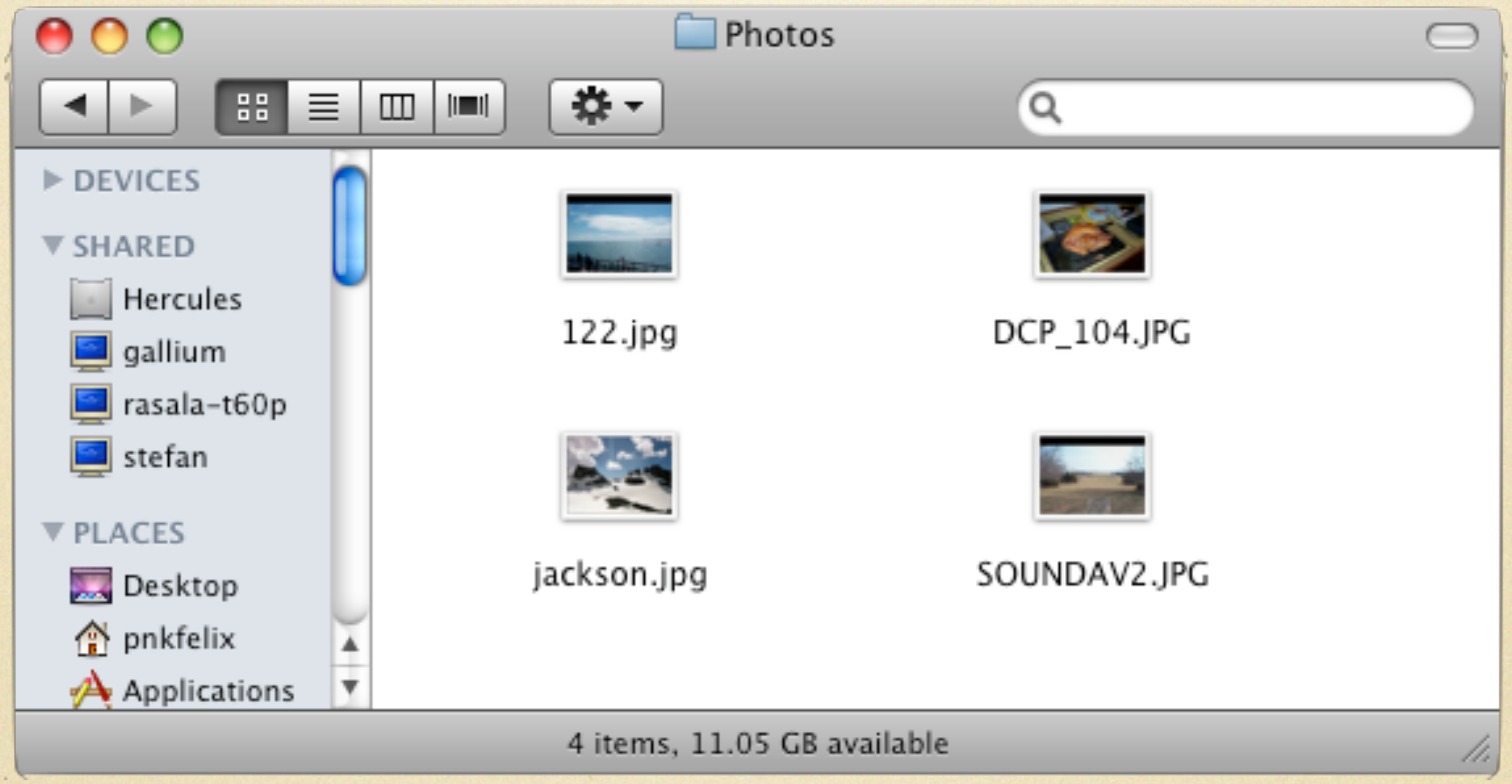
Linux (Intel x86)




```
> (list-directory ".")
("." ".." "122.jpg" "DCP_104.JPG"
 "jackson.jpg" "SOUNDAV2.JPG")
```


Felix tried the code on Linux. Lo and behold, it worked!

Mac OS X (Intel x86)




```
> (list-directory ".")
("" "" ".jpg" "_104.JPG"
 "kson.jpg" "NDAV2.JPG")
```


Encouraged, Felix tried the code on Mac OS X. Here’s the result.

The Investigation

```
;; list-directory : String -> [Listof String]
(define (list-directory dirname)
  (let ((dir (unix/opendir dirname)))
    (if (zero? dir)
        (error 'list-directory path)
        (let loop ((files '()))
          (let ((ent (unix/readdir dir)))
            (if (zero? ent)
                (begin (unix/closedir dir)
                       (reverse files))
                (loop (cons (dirent->name ent)
                            files))))))))
```


So Felix started looking at the code.
list-directory is a pretty standard function. (Easy to port from C for loop in example code.)
It just opens the directory, iterates through its entries (building up a list of names from each), and closes the directory.

The Investigation

```
;; list-directory : String -> [Listof String]
(define (list-directory dirname)
  (let ((dir (unix/opendir dirname)))
    (if (zero? dir)
        (error 'list-directory path)
        (let loop ((files '()))
          (let ((ent (unix/readdir dir)))
            (if (zero? ent)
                (begin (unix/closedir dir)
                       (reverse files))
                (loop (cons (dirent->name ent)
                      files))))))))
```


32

If somethings wrong, it seems like it would be something in unix/opendir, unix/readdir, unix/closedir, or
dirent->name

The Culprit

```
;; The offsets in the dirent accessors
;; are probably x86-Linux-specific!!

(define unix/opendir
  (foreign-procedure "opendir" '(string) 'uint))
(define unix/readdir
  (foreign-procedure "readdir" '(uint) 'uint))
(define unix/closedir
  (foreign-procedure "closedir" '(uint) 'int))

(define (dirent->name ent)
  (%peek-string (+ ent 11)))
```


So Felix inspects the definitions of those functions, elsewhere in the file.
And there we find the problem (pointed out directly in Lars's comments).

The uints are dirent memory addresses; the int is an error return code.
(To learn more about the interface to foreign-procedure, see the paper.)

The Culprit

```
;; The offsets in the dirent accessors
;; are probably x86-Linux-specific!!

(define unix/opendir
  (foreign-procedure "opendir" '(string) 'uint))
(define unix/readdir
  (foreign-procedure "readdir" '(uint) 'uint))
(define unix/closedir
  (foreign-procedure "closedir" '(uint) 'int))

(define (dirent->name ent)
  (%peek-string (+ ent 11)))
```


34

%peek-string is an (unsafe) memory accessor in Larceny.
Here's we're just calculating the memory address of the entry's name by adding 11 to the entry's start.

"Portable" code

- Referring to field names is portable, but not field offsets
- On UNIX, "`struct dirent`" must have a "`d_name`" field holding filename characters
- On our Linux system, "`d_name`" is at offset 11
- But not on Mac OS X

“Portable” code

- One solution:

  1. Transcribe desired structure definition into Scheme special forms

  2. Macro-expand forms to offsets according to target’s ABI

“Portable” code

- One solution:

1. Transcribe desired structure definition into Scheme special forms

2. Macro-expand forms to offsets according to target's ABI

37

“Portable” code

- Library developers (often) have freedom to extend structure definitions with new fields
    - E.g. “`struct dirent`” definitions differ
- Transcribed structure definitions are *not portable*

“Portable” code

- Portability requires extracting information from the header file
  - Did not want to write a C parser
  - Insight: generating C programs that extract info (e.g. field offsets) is *much easier* task
- Procedural macros allow one to generate, compile, and execute such a program statically!

Mention that the trick is also used by Haskell FFI.
(and GNU configure set a precedent for generating small C pgms to reflect on the host system.)
BUT we’re doing it as a macro!

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```


I will now illustrate the idea behind the define-c-info syntax by showing how its expansion works in one case.
Here we interface to a C pair struct, and we want access to two of its fields.

define-c-info form

```
(define-c-info (include "pair.h")
  (struct "pair" (x-offs "x") (y-offs "y")))
```

```
;; pair.h
struct pair {
  int id;
  int x;
  char c;
  int y;
};
```

```
(begin
  (define x-offs 4)
  (define y-offs 12))
```


I will now illustrate the idea behind the define-c-info syntax by showing how its expansion works in one case.
Here we interface to a C pair struct, and we want access to two of its fields.

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```


42

I will now illustrate the idea behind the define-c-info syntax by showing how its expansion works in one case.
Here we interface to a C pair struct, and we want access to two of its fields.

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
#include “pair.h”
#include <stdio.h>
int main() {
 printf(“\n(\n”);
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.x-
                      (char*)&s))); }
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.y-
                      (char*)&s))); }
 printf(“\n)\n”); return 0;
}
```

1. generate C code

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
#include “pair.h”
#include <stdio.h>
int main() {
 printf(“\n(\n”);
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.x-
                      (char*)&s))); }
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.y-
                      (char*)&s))); }
 printf(“\n)\n”); return 0;
}
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
#include “pair.h”
#include <stdio.h>
int main() {
 printf(“\n(\n”);
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.x-
                        (char*)&s))); }
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.y-
                        (char*)&s))); }
 printf(“\n)\n”); return 0;
}
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
#include “pair.h”
#include <stdio.h>
int main() {
 printf(“\n(\n”);
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.x-
                      (char*)&s))); }
 { struct pair s;
   printf(“%ld ”,((long)((char*)&s.y-
                      (char*)&s))); }
 printf(“\n)\n”); return 0;
}
```

2. compile to `a.out`

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

44

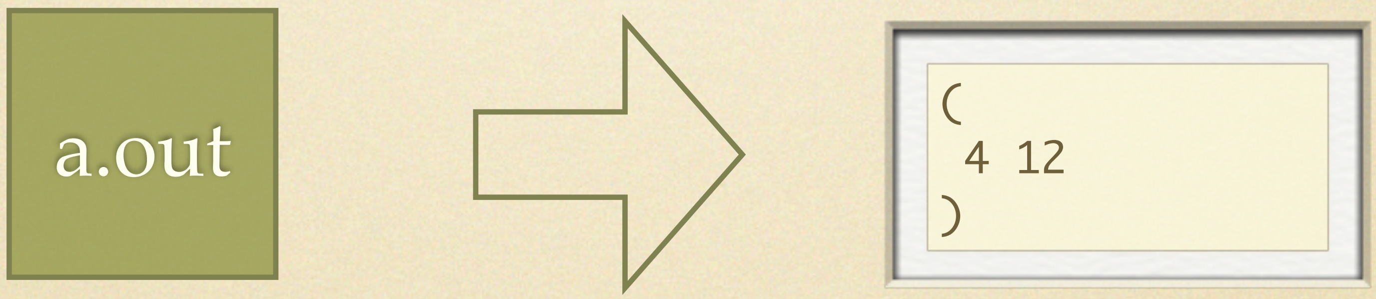
define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

45

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```



3. run `a.out`, piping results to temp file

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
(
 4 12
)
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
(
 4 12
)
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
(
 4 12
)
```

```
(begin
  (define x-offs 4)
  (define y-offs 12))
```

4. read temp file and generate binding form

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
(begin
  (define x-offs 4)
  (define y-offs 12))
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
;; pair.h
struct pair {
  int id;
  int x;
  char c;
  int y;
};
```

```
(begin
  (define x-offs 4)
  (define y-offs 12))
```

define-c-info form

```
(define-c-info (include “pair.h”)
  (struct “pair” (x-offs “x”) (y-offs “y”)))
```

```
;; pair.h
struct pair {
  int id;
  int x;
  char c;
  int y;
};
```

```
(begin
  (define x-offs 4)
  (define y-offs 12))
```

(Actual numbers depend on contents of “pair.h”)

47

Fixed/Portable UNIX Filesystem Interface

```
(define (dirent->name ent)
  (%peek-string (+ ent 11)))

(define (dirent->name ent)
  (define-c-info (include<> "dirent.h")
    (struct "dirent" (name-offs "d_name")))
  (%peek-string (+ ent name-offs)))
```

- This version works on Linux, Mac OS X, Solaris
- Windows: same idea, but different API

More High Level Interface Syntax

- `define-c-info` has proven to be a useful core construct, though low-level
- Foundation for other syntax
  - `define-c-struct`
  - `define-c-enum`, `define-c-enum-set`

49

see paper for details on the other macros

Related Work: FFI's for Scheme

- (vast amount of Lisp FFI material)
- `esh` [Rose and Muller, 1992]: tight integration, tight constraints
- SRFI-50 [Kelsey and Sperber, 2003]: client writes glue in C
- PLT Scheme [Barzilay and Orlovsky, 2004]: "stay in the fun world"

I only begin to skim the surface of interfacing Lisp/C in the related work section.

Related Work: interface extraction

- `esh` [Rose and Muller, 1992]: maps headers to UNIX object files
- `SWIG` [Beazley, 1996]: processes subset of C and C++ into scripting language
- `FIG` [Reppy and Song, 2006]: process header files via a declarative DSL

esh: C macros become Scheme functions!
SWIG: you have you write your interface in a C-like language; it won't handle arbitrary headers directly
FIG: combines declarative DSL and term-rewriting to derive interfaces to foreign libraries

Conclusion

- Larceny's low-level FFI structure
  - largely orthogonal to Larceny VM design
  - not simple; but much complexity is kept in Scheme code, not C code
- Larceny's high-level FFI functionality: *simple* macros that process header files

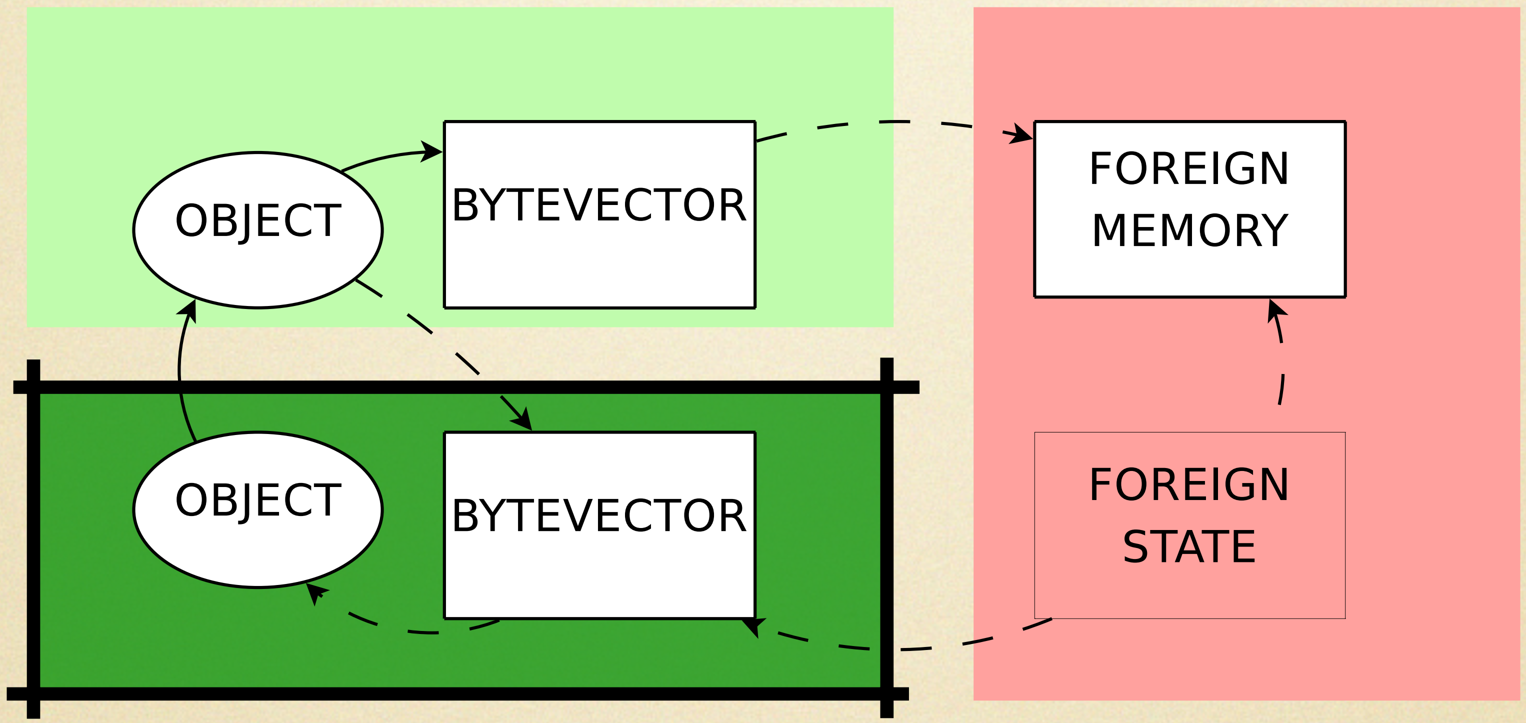
thanks!

Low Level Heap Structure

(how to satisfy the garbage collector)

I’ve shown control flow details so far; now I’m going to show how the objects implementing that control are distributed throughout the heap.
But first I need to explain the diagram conventions.

Heap Diagram legend

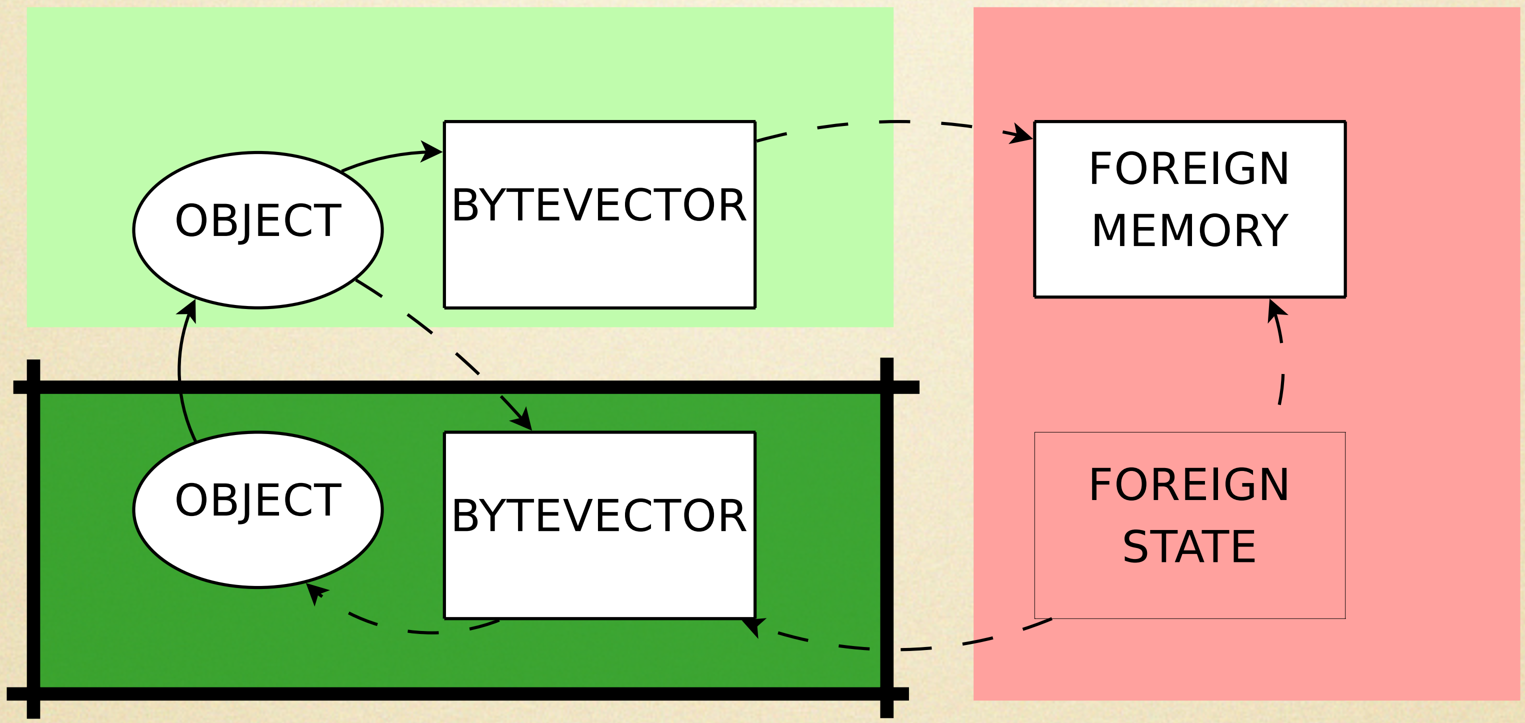


- Ovals: GC traced objects
- Rectangles: untraced objects / memory / code

56

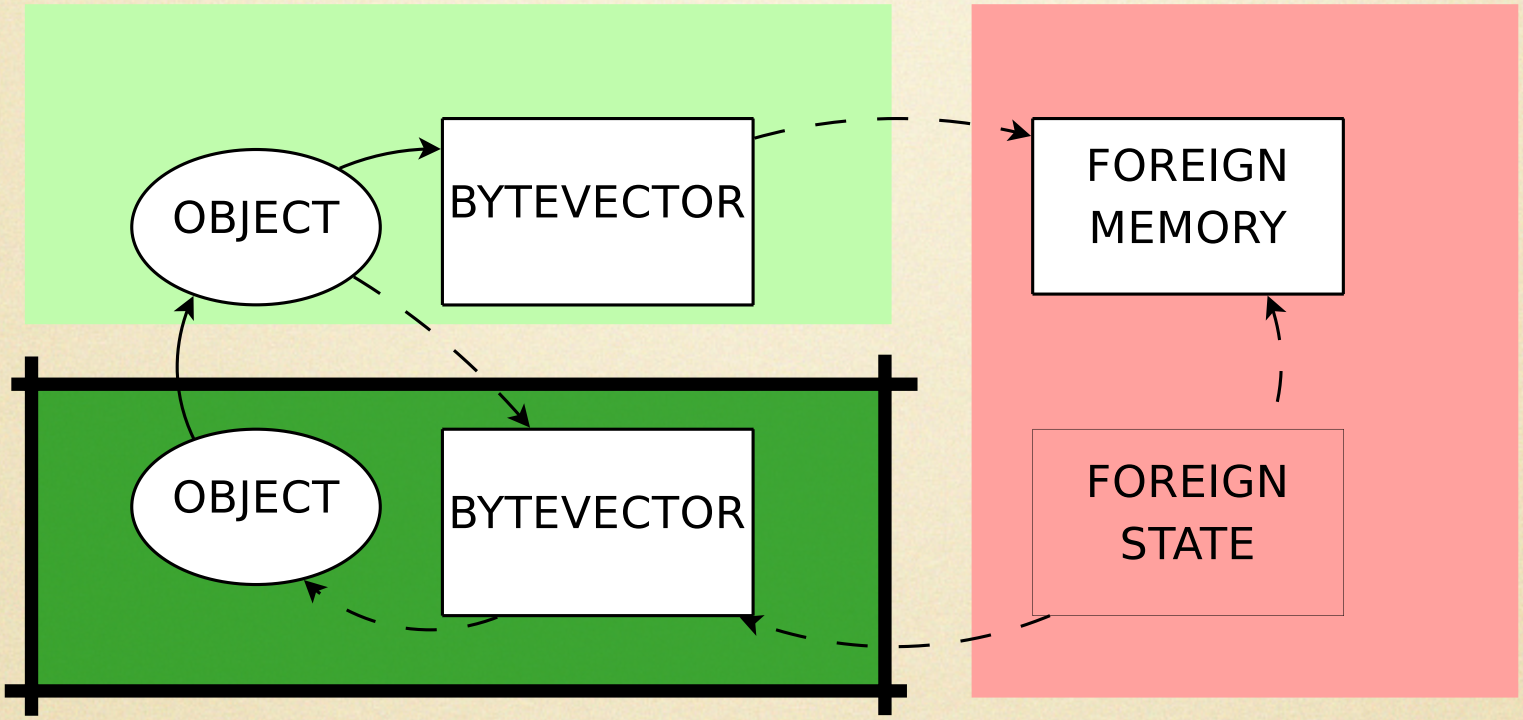
Ovals are e.g. closures, pairs. Rectangles are e.g. machine code, C runtime fcns, Scheme string contents
“Foreign State” in lower right corner is a fuzzy abstraction of the C memory state

Heap Diagram legend



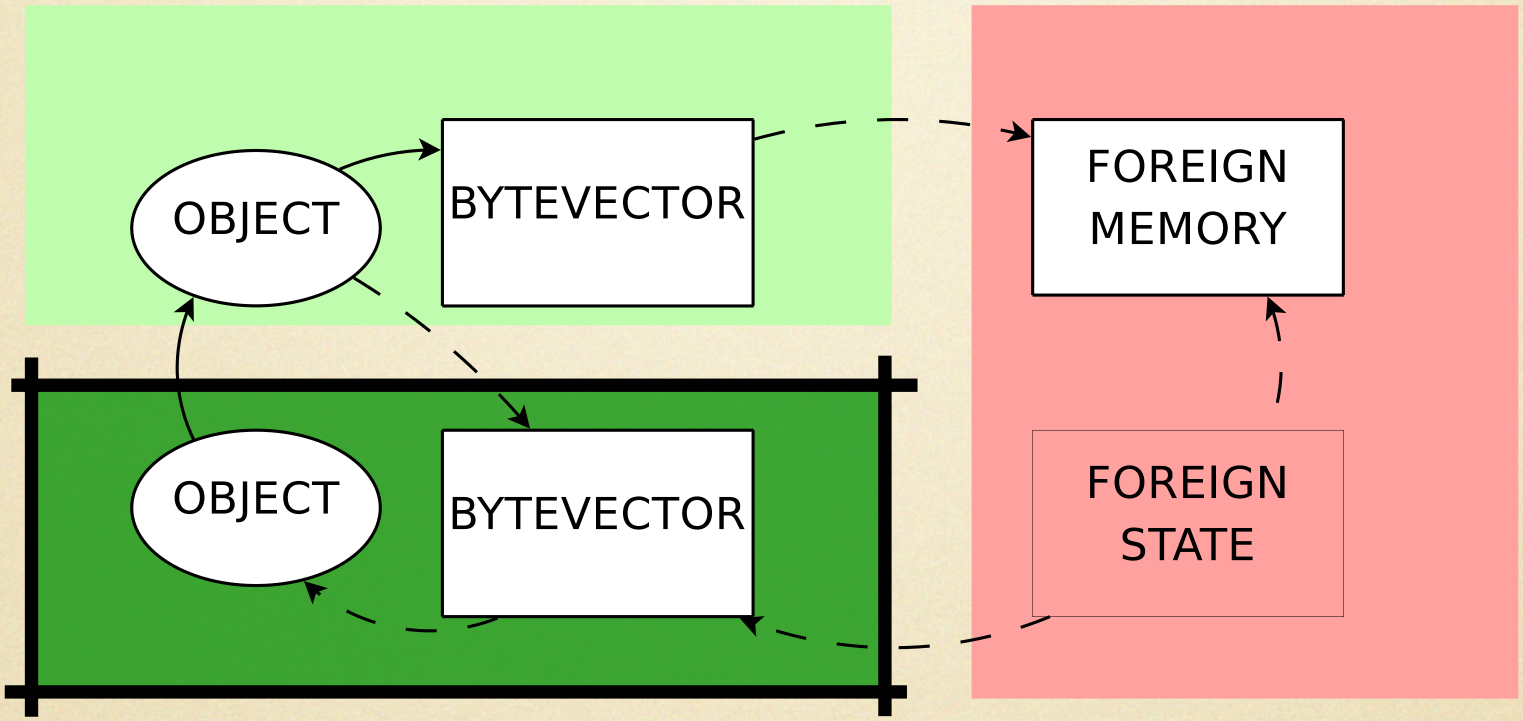
- Solid arrows: GC traced references
- Dashed arrows: untraced (encoded) addresses

Heap Diagram legend



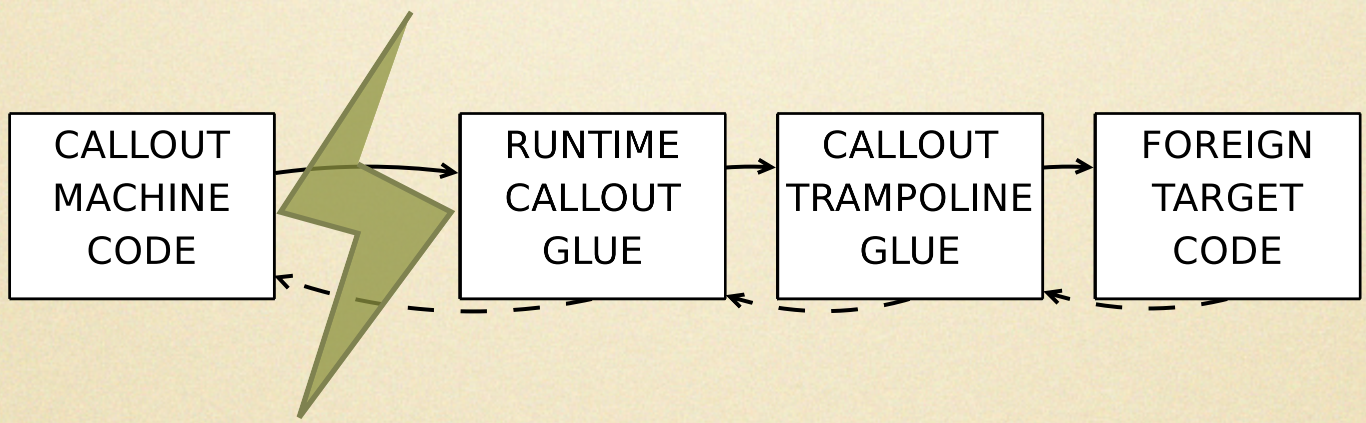
- Scheme managed versus foreign memory
- non-moving vs relocatable memory

Heap Diagram rules



- Solid arrows only originate at ovals
- Dashed arrows cannot point to relocatable
- Solid arrows cannot point to C runtime state

Heap Structure: Callouts

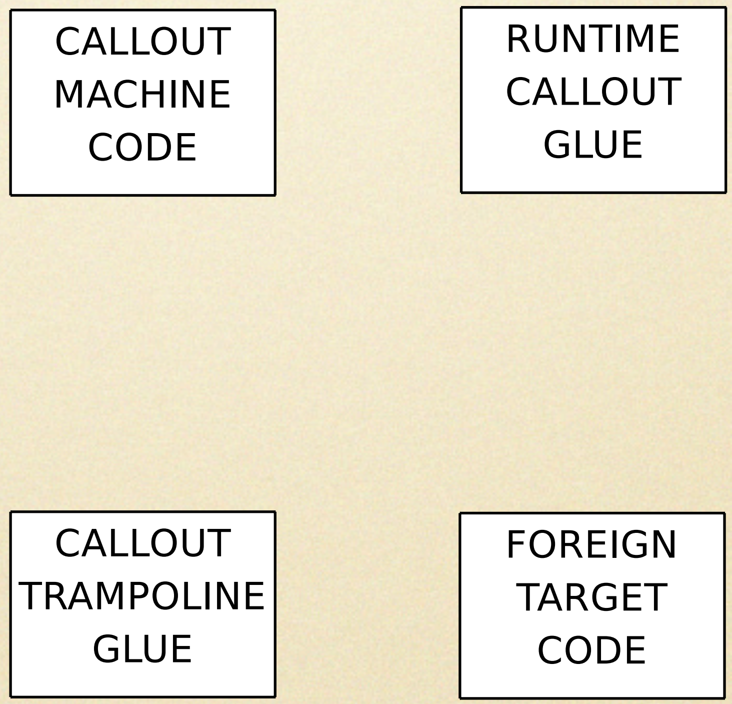


CALLOUT CONTROL FLOW

60

We start off with the Runtime functions and Foreign Target;
we get to create the MacScheme stuff.
(keep in mind that final control flow is going to follow the “Z”)
... get to start by creating a Trampoline object

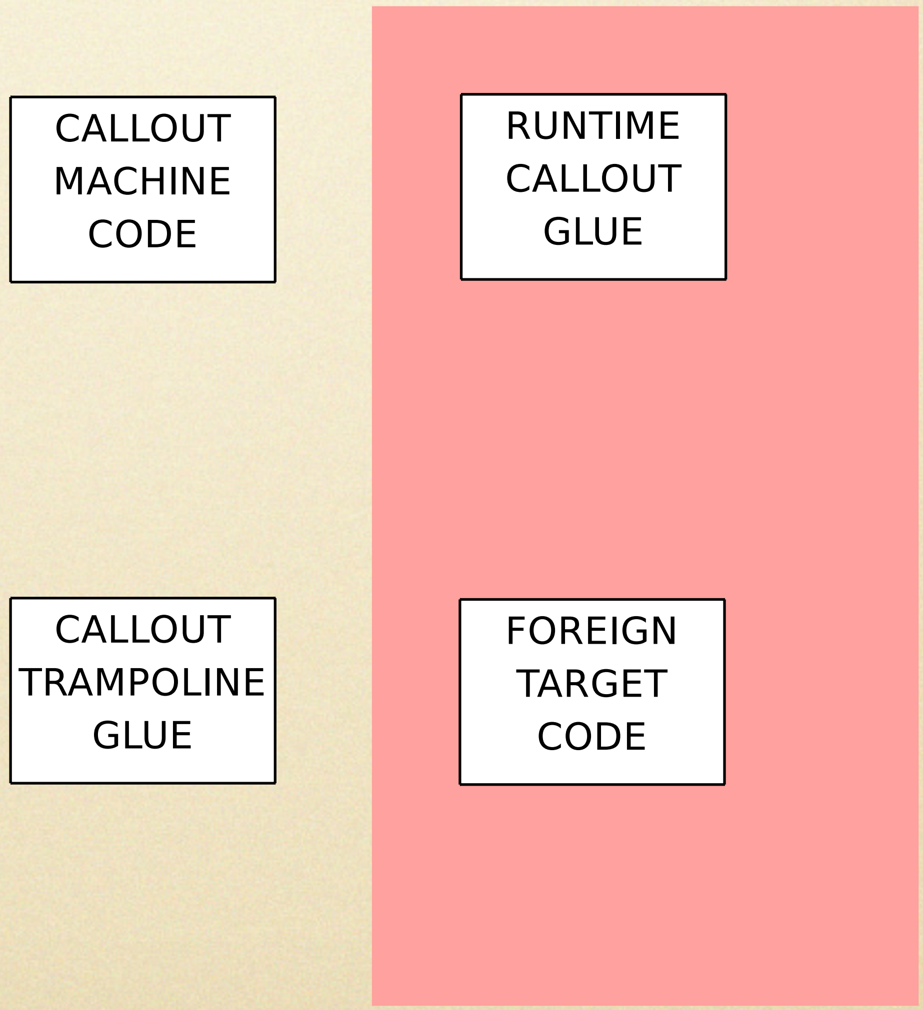
Heap Structure: Callouts



60

We start off with the Runtime functions and Foreign Target;
we get to create the MacScheme stuff.
(keep in mind that final control flow is going to follow the “Z”)
... get to start by creating a Trampoline object

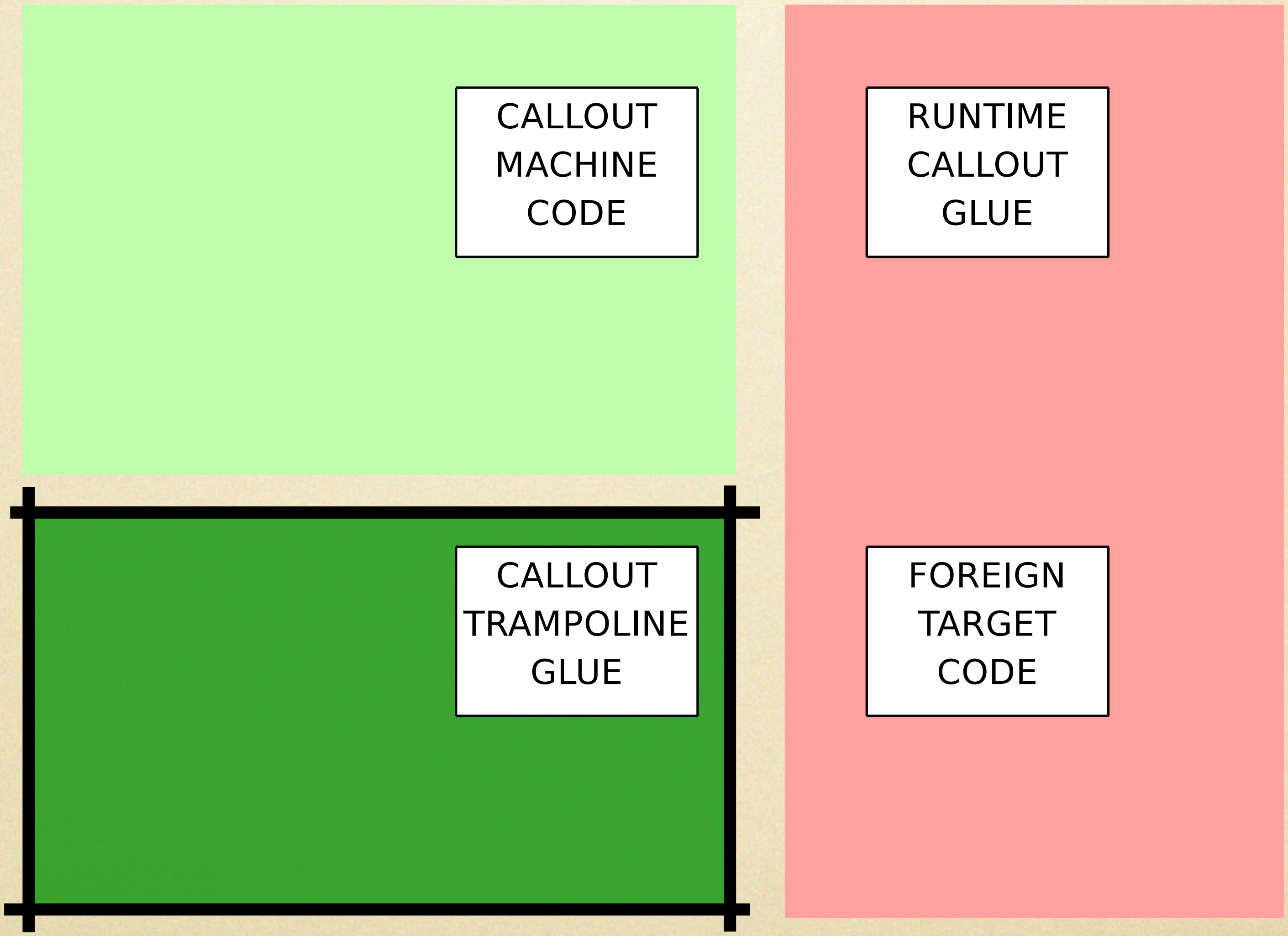
Heap Structure: Callouts



60

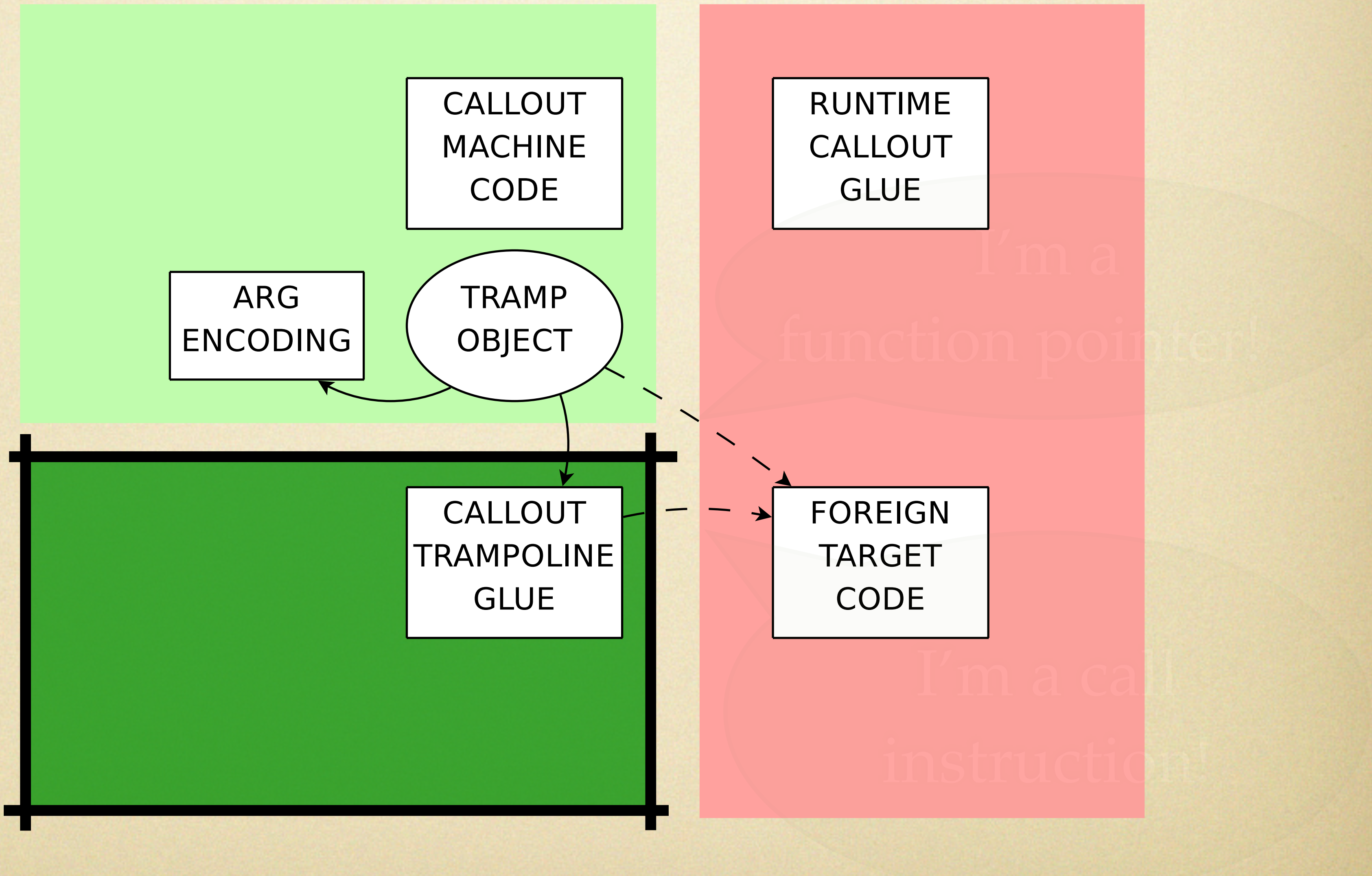
We start off with the Runtime functions and Foreign Target;
we get to create the MacScheme stuff.
(keep in mind that final control flow is going to follow the “Z”)
... get to start by creating a Trampoline object

Heap Structure: Callouts

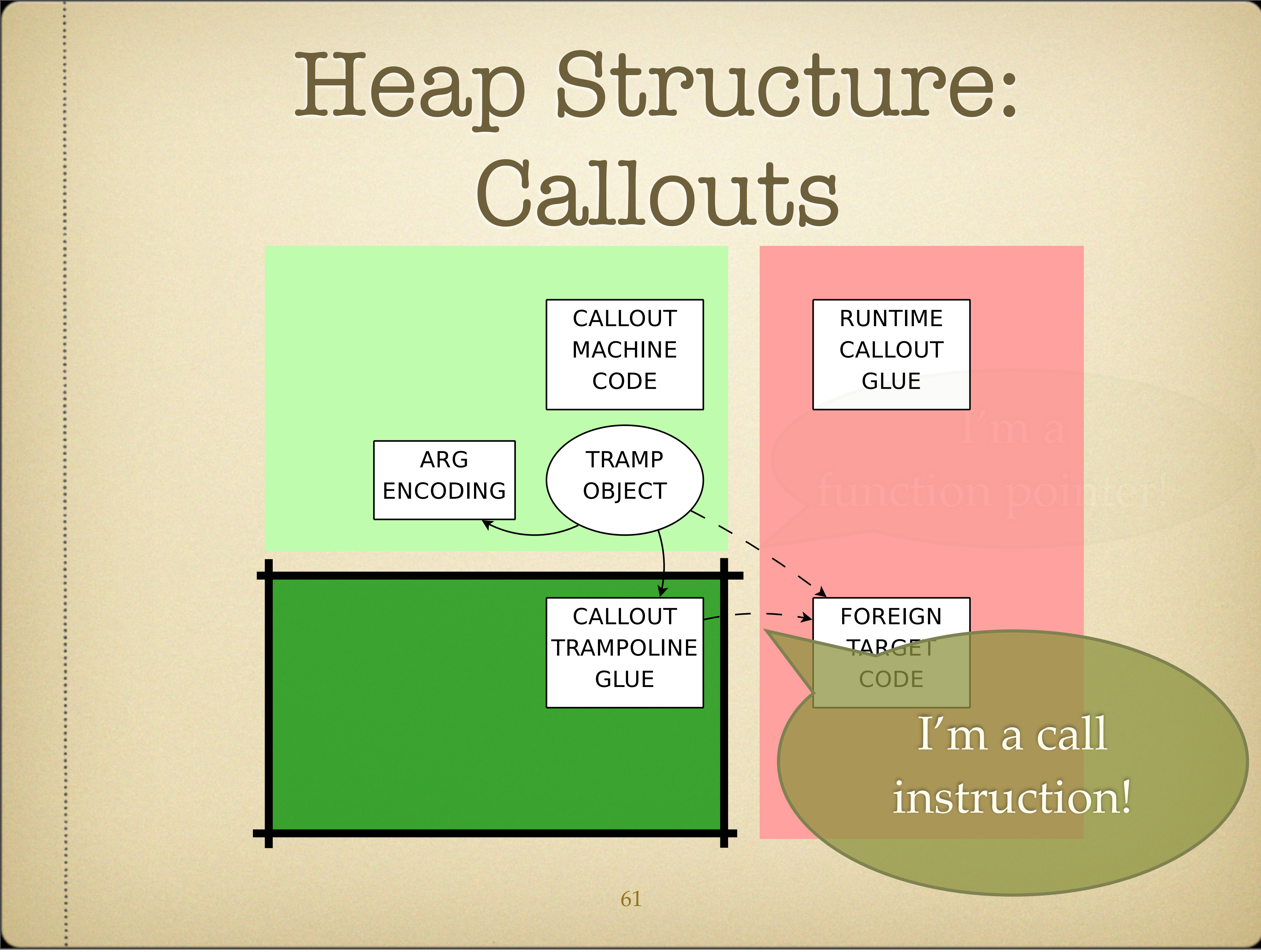


We start off with the Runtime functions and Foreign Target;
we get to create the MacScheme stuff.
(keep in mind that final control flow is going to follow the “Z”)
... get to start by creating a Trampoline object

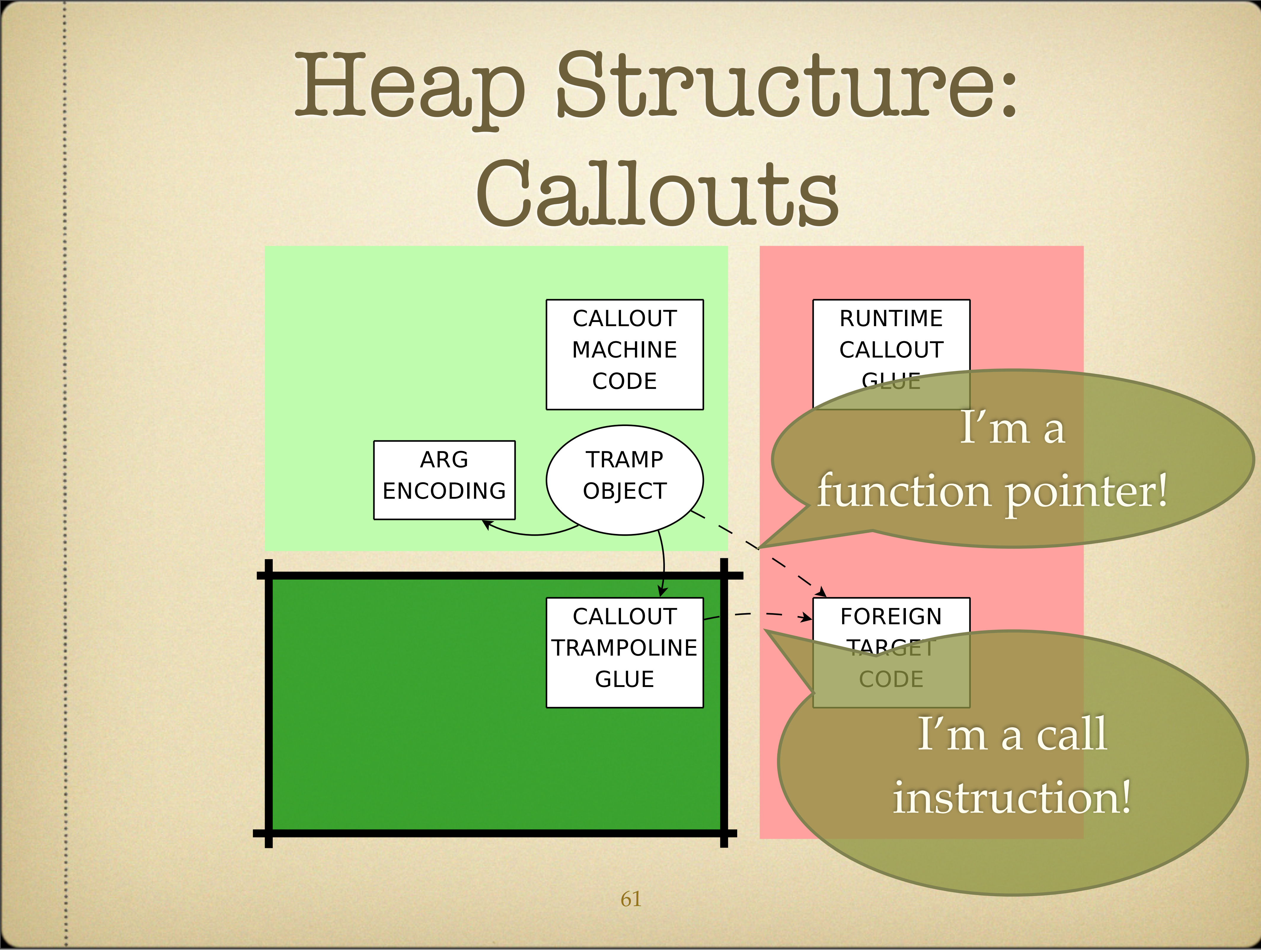
Heap Structure: Callouts



Q: What are these dashed arrows?
... this Tramp Object just represents the target; its *not* what Scheme client code directly invokes.

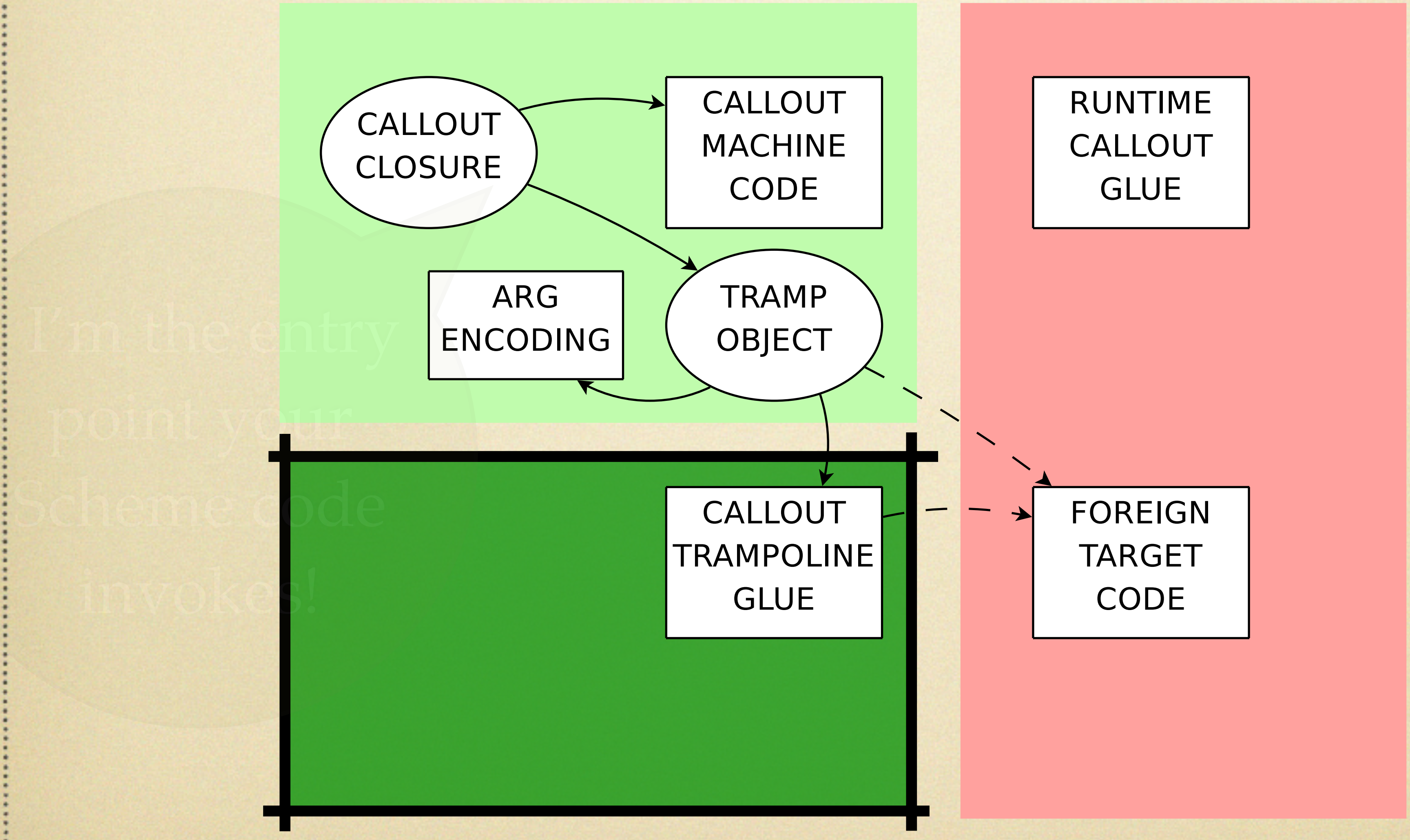


Q: What are these dashed arrows?
... this Tramp Object just represents the target; its *not* what Scheme client code directly invokes.



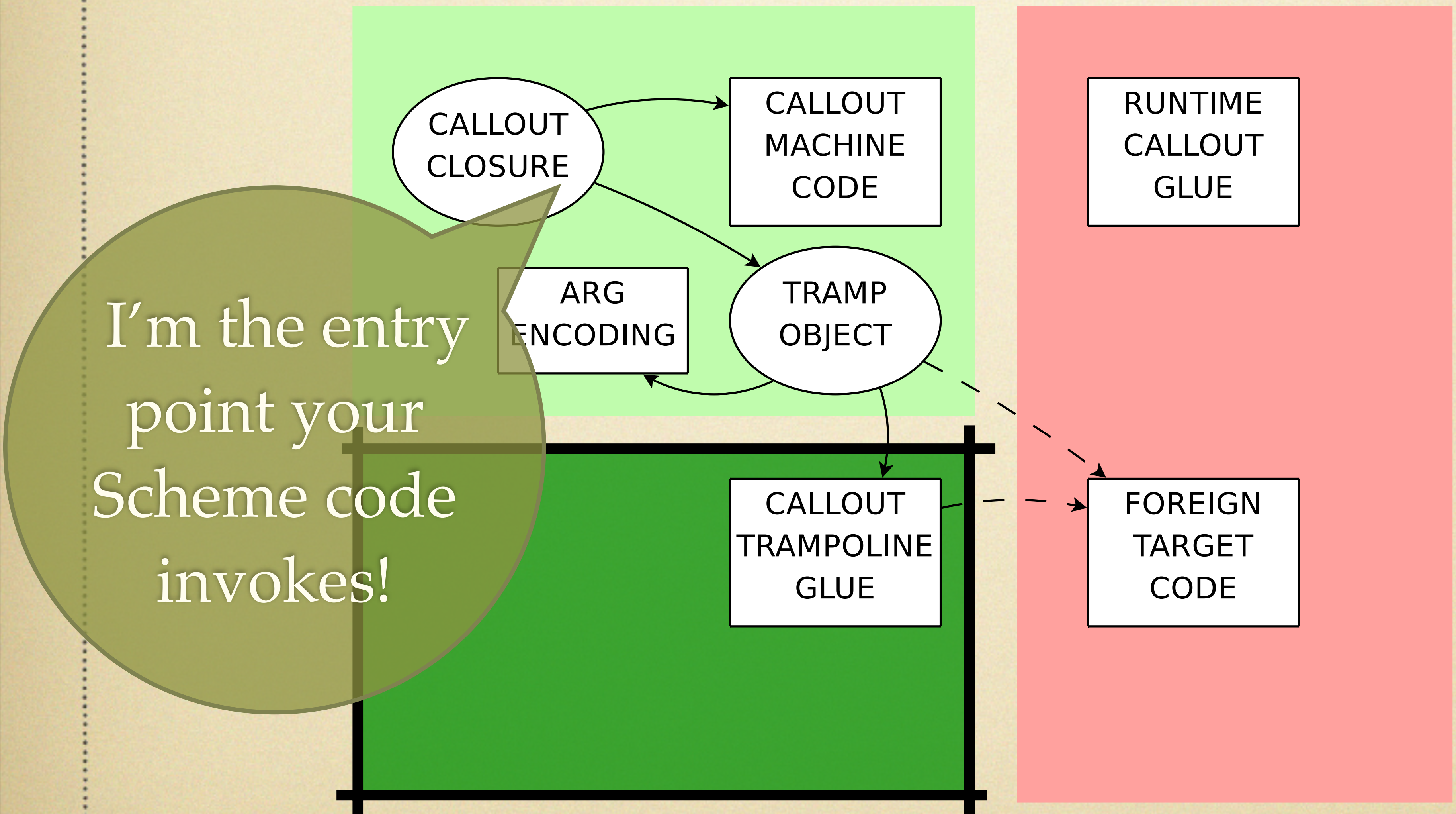
Q: What are these dashed arrows?
... this Tramp Object just represents the target; its *not* what Scheme client code directly invokes.

Heap Structure: Callouts



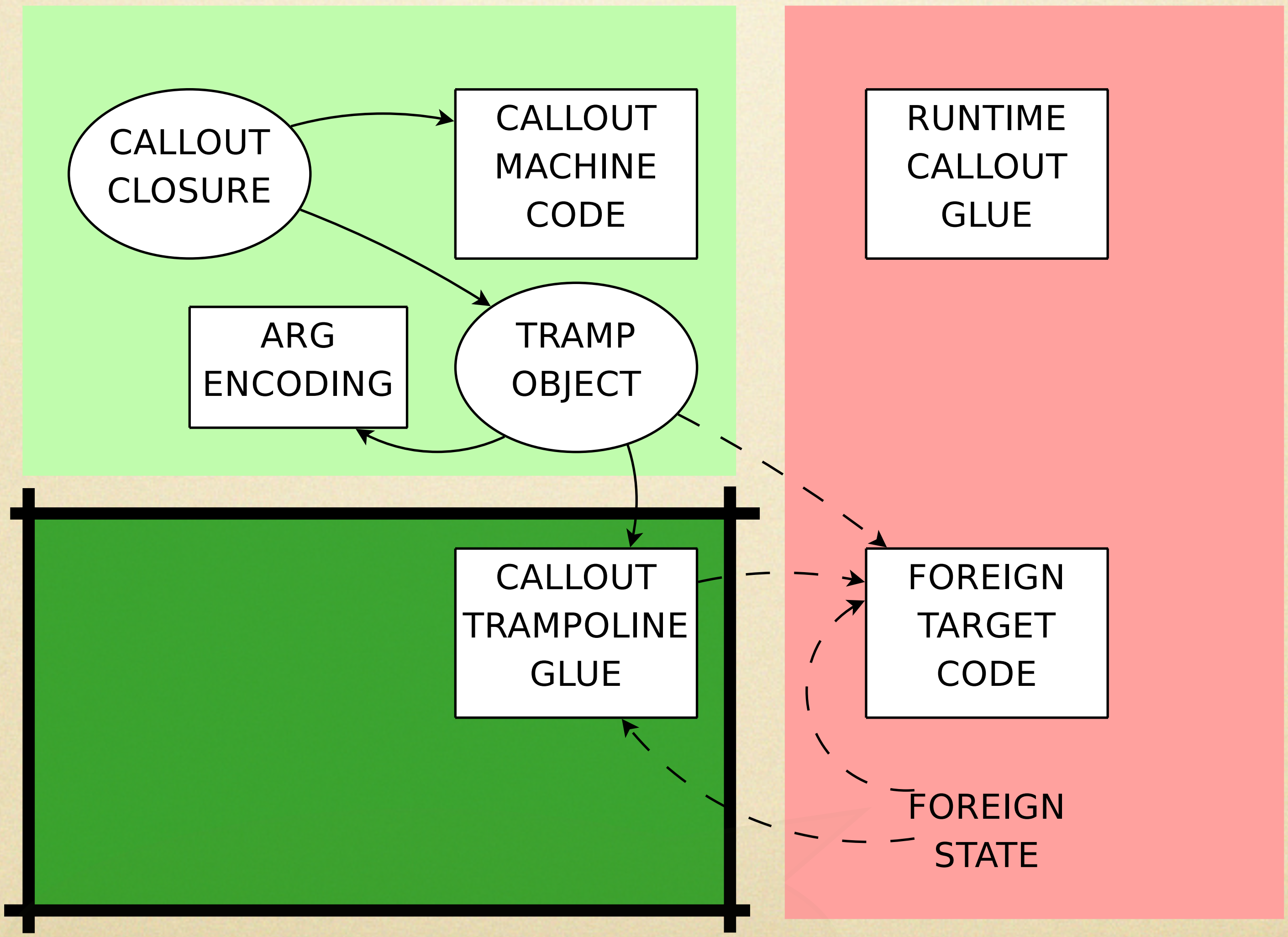
This is the closure that client code invokes. It extracts the ARG ENCODING and TRAMP GLUE, and passes them along to the runtime callout.
(... but there's one last detail: C function invocations push return addresses onto the C stack...)

Heap Structure: Callouts



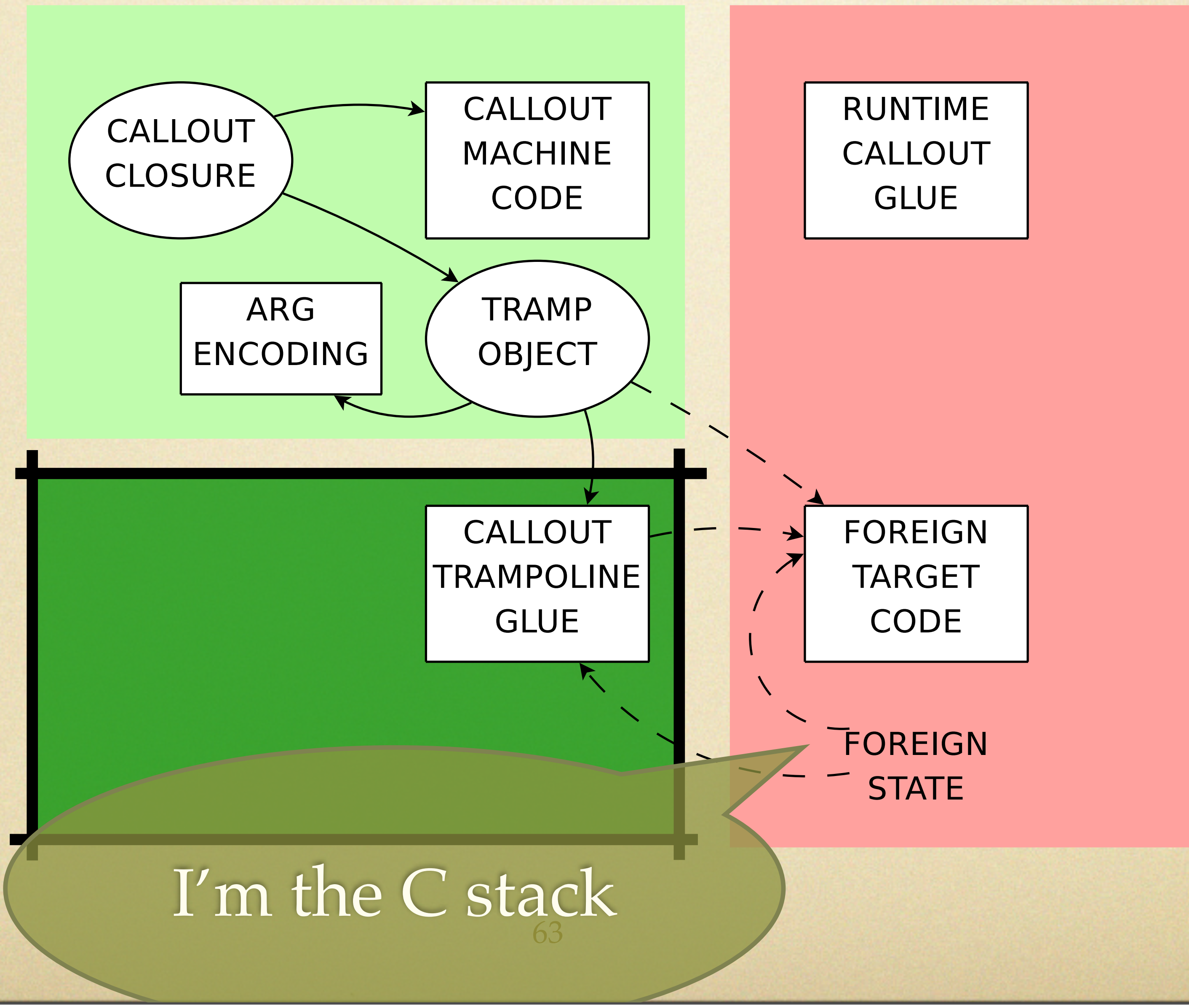
This is the closure that client code invokes. It extracts the ARG ENCODING and TRAMP GLUE, and passes them along to the runtime callout.
(... but there's one last detail: C function invocations push return addresses onto the C stack...)

Heap Structure: Callouts



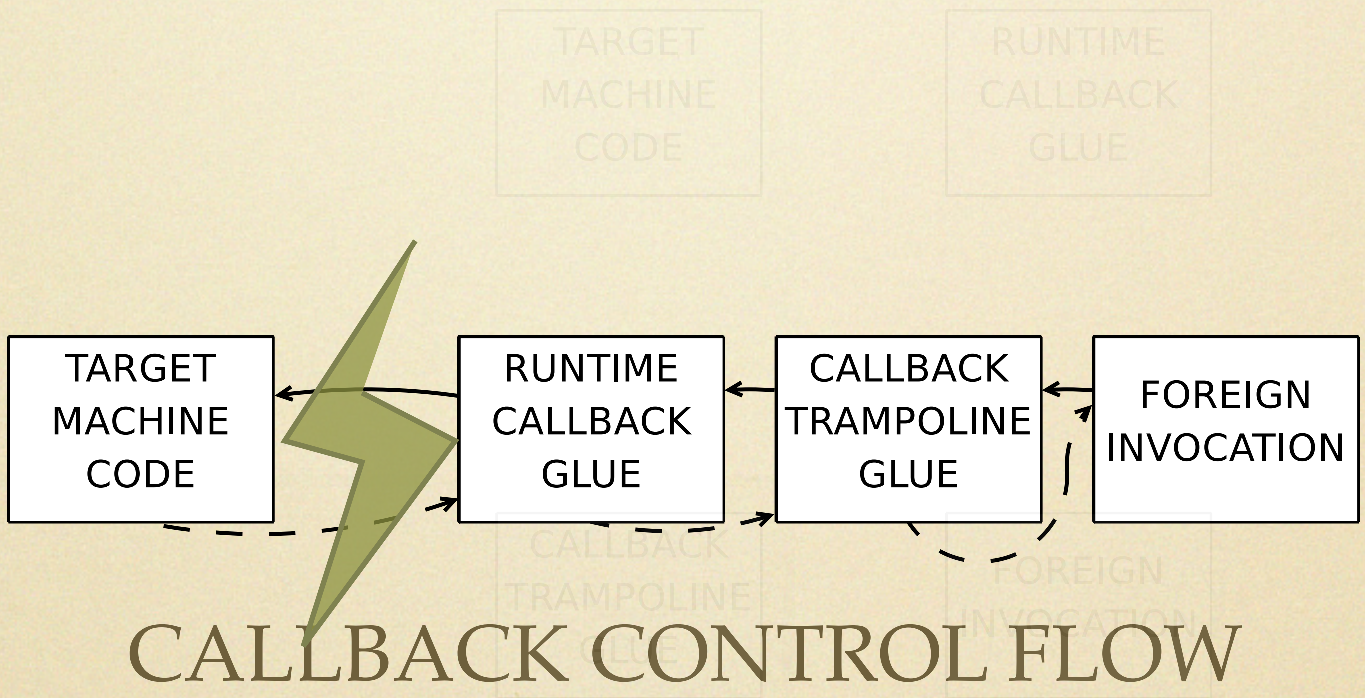
So to properly represent the references from the foreign memory, we add these dashed arrows.
and with this, we have an accurate diagram that also satisfies all of the rules.
Q: to ponder: does it actually matter that a reference into the Trampoline Glue is on the C stack? When/how could the GC be invoked while its on the stack?

Heap Structure: Callouts



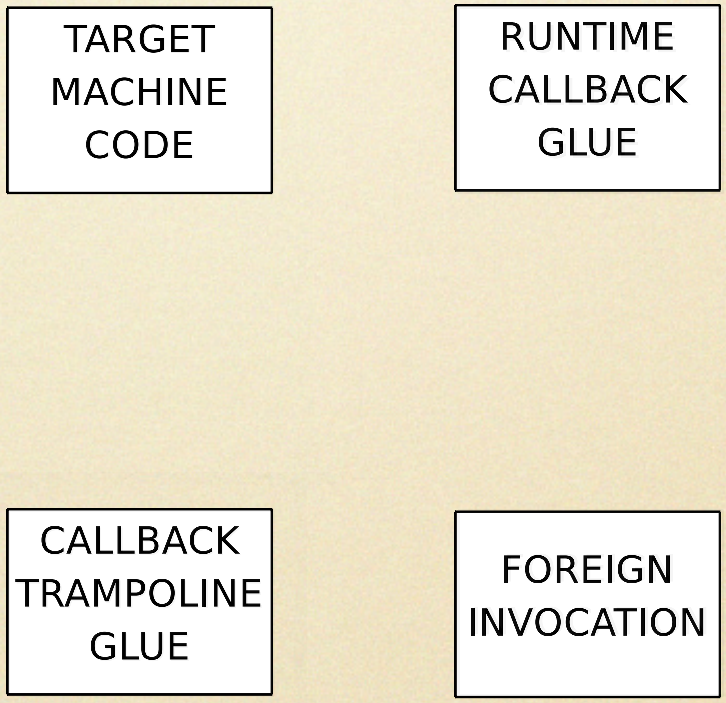
So to properly represent the references from the foreign memory, we add these dashed arrows.
and with this, we have an accurate diagram that also satisfies all of the rules.
Q: to ponder: does it actually matter that a reference into the Trampoline Glue is on the C stack?
When/how could the GC be invoked while its on the stack?

Heap Structure: Callbacks



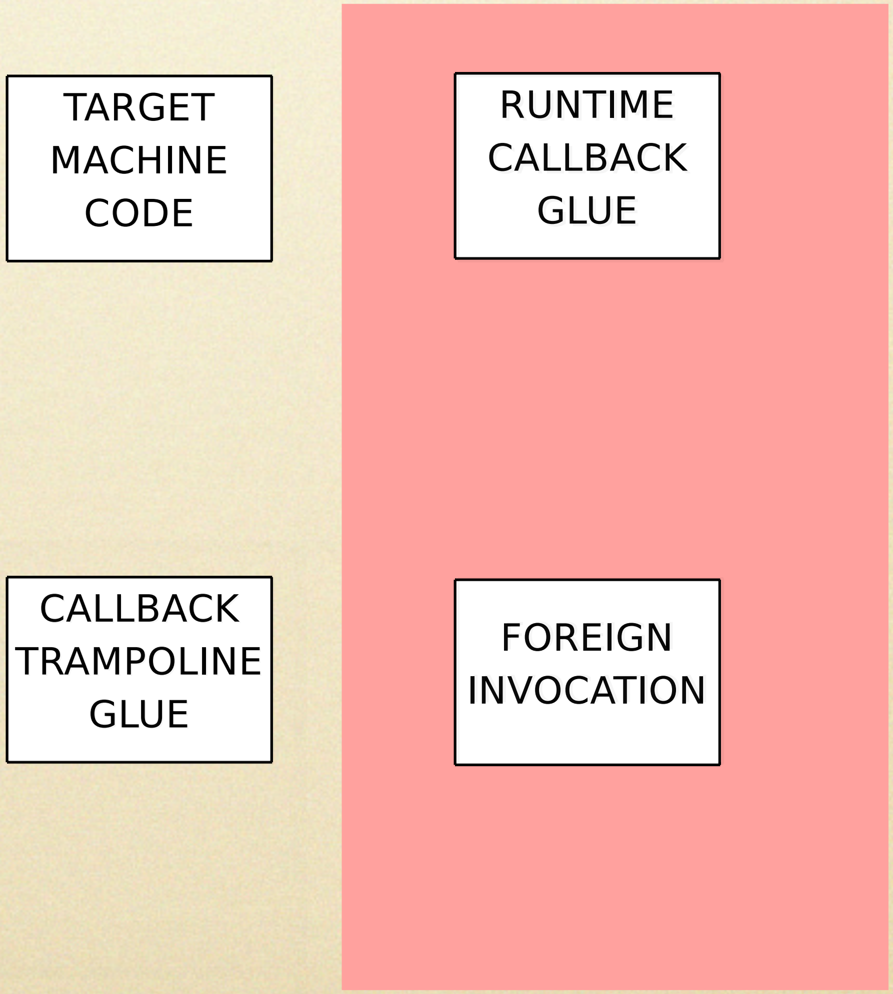
Distributing control amongst heap objects.
Remember: we start at the foreign invocation, go through trampoline, then runtime, and finally hit the target closure (the “Z” in reverse).

Heap Structure: Callbacks



Distributing control amongst heap objects.
Remember: we start at the foreign invocation, go through trampoline, then runtime, and finally hit the target closure (the “Z” in reverse).

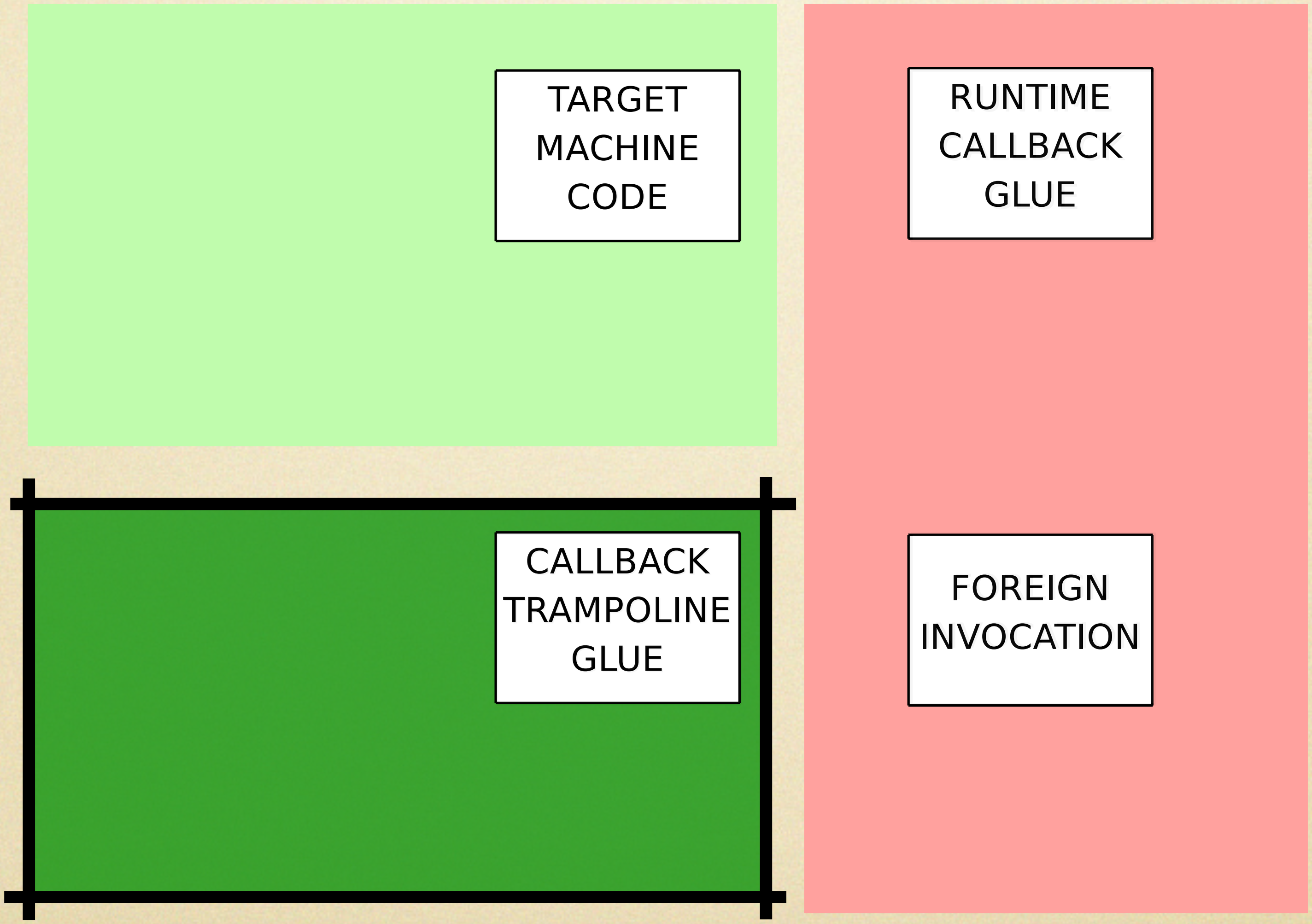
Heap Structure: Callbacks



64

Distributing control amongst heap objects.
Remember: we start at the foreign invocation, go through trampoline, then runtime, and finally hit the target closure (the "Z" in reverse).

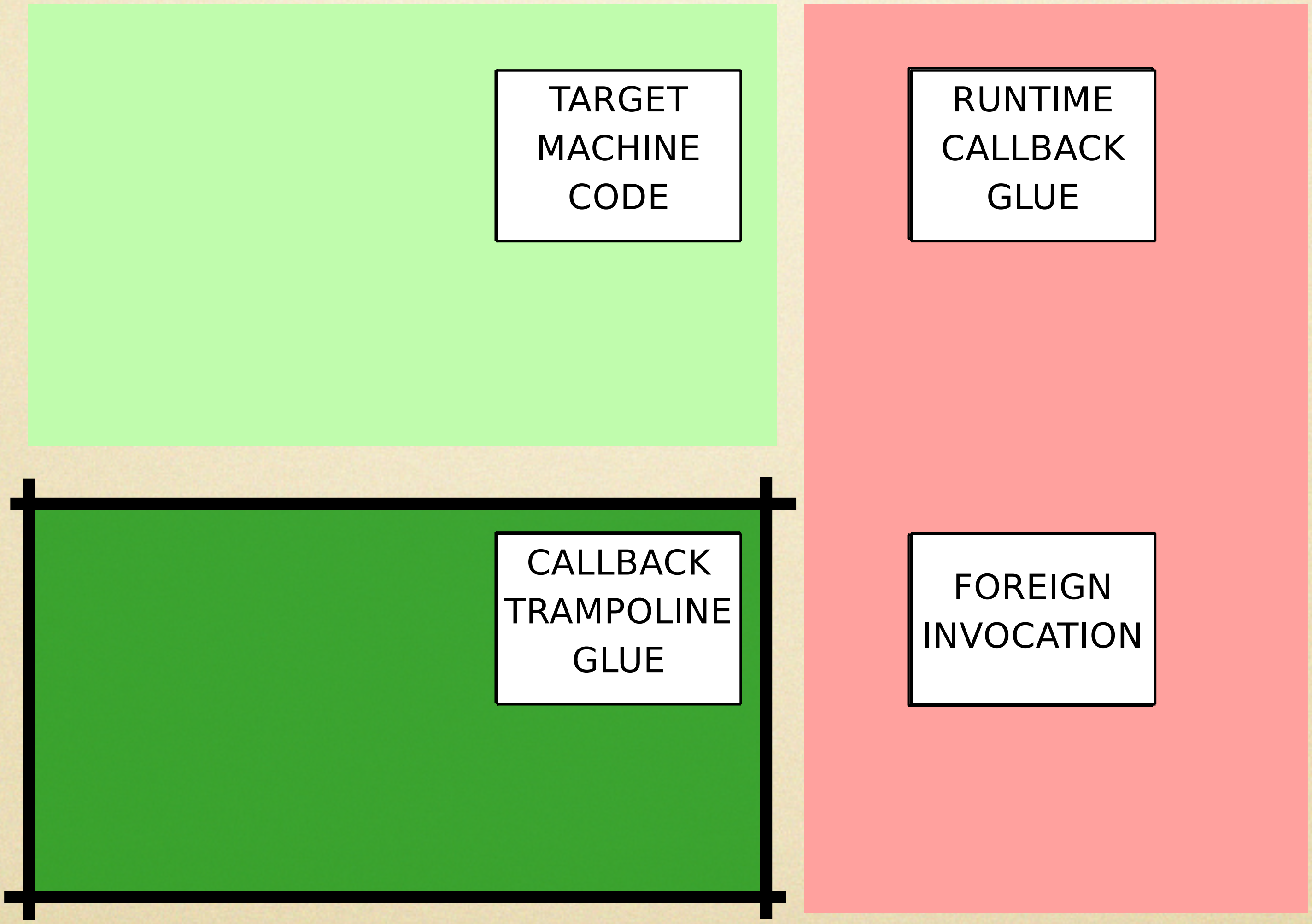
Heap Structure: Callbacks



64

Distributing control amongst heap objects.
Remember: we start at the foreign invocation, go through trampoline, then runtime, and finally hit the target closure (the “Z” in reverse).

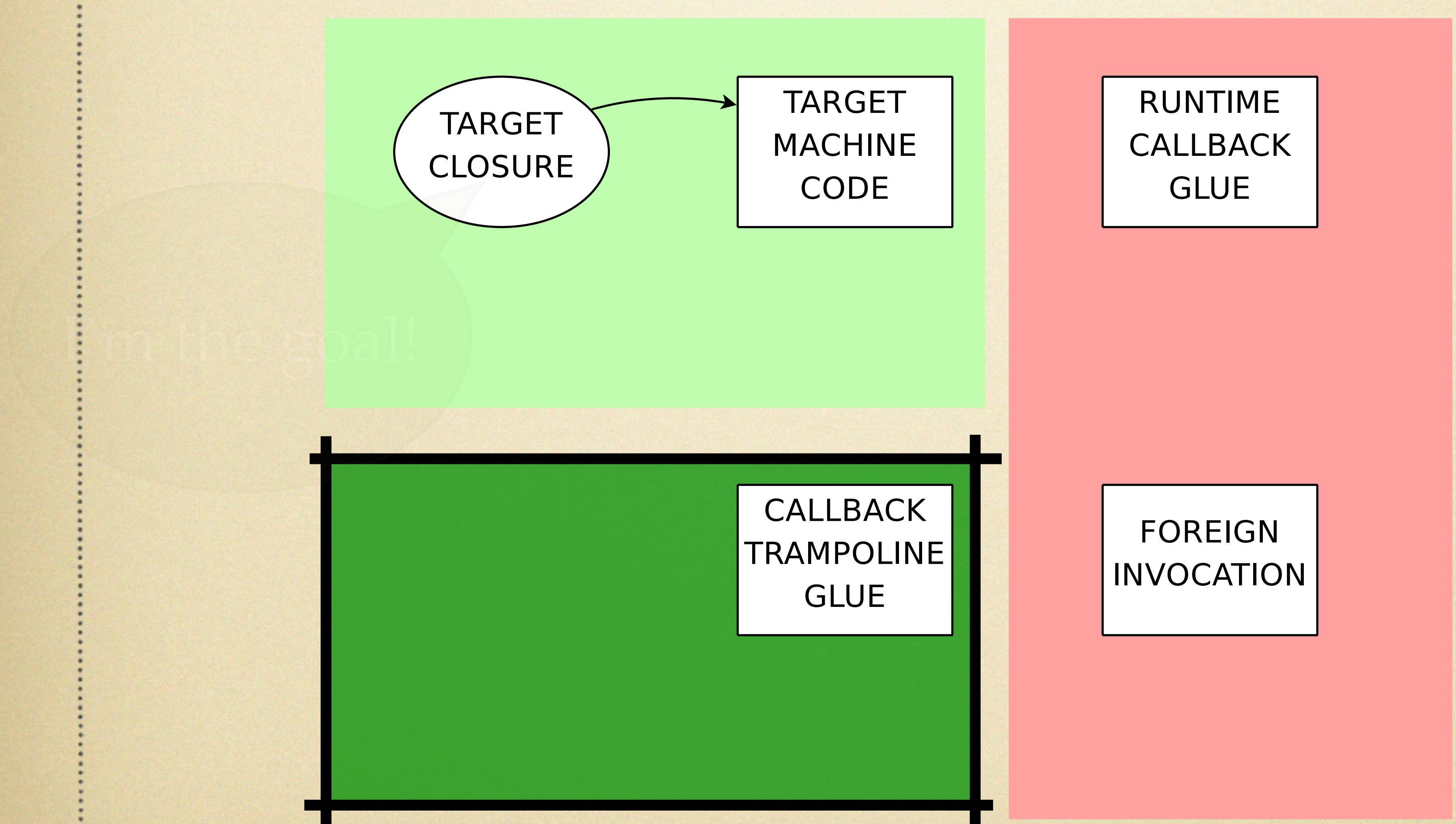
Heap Structure: Callbacks



64

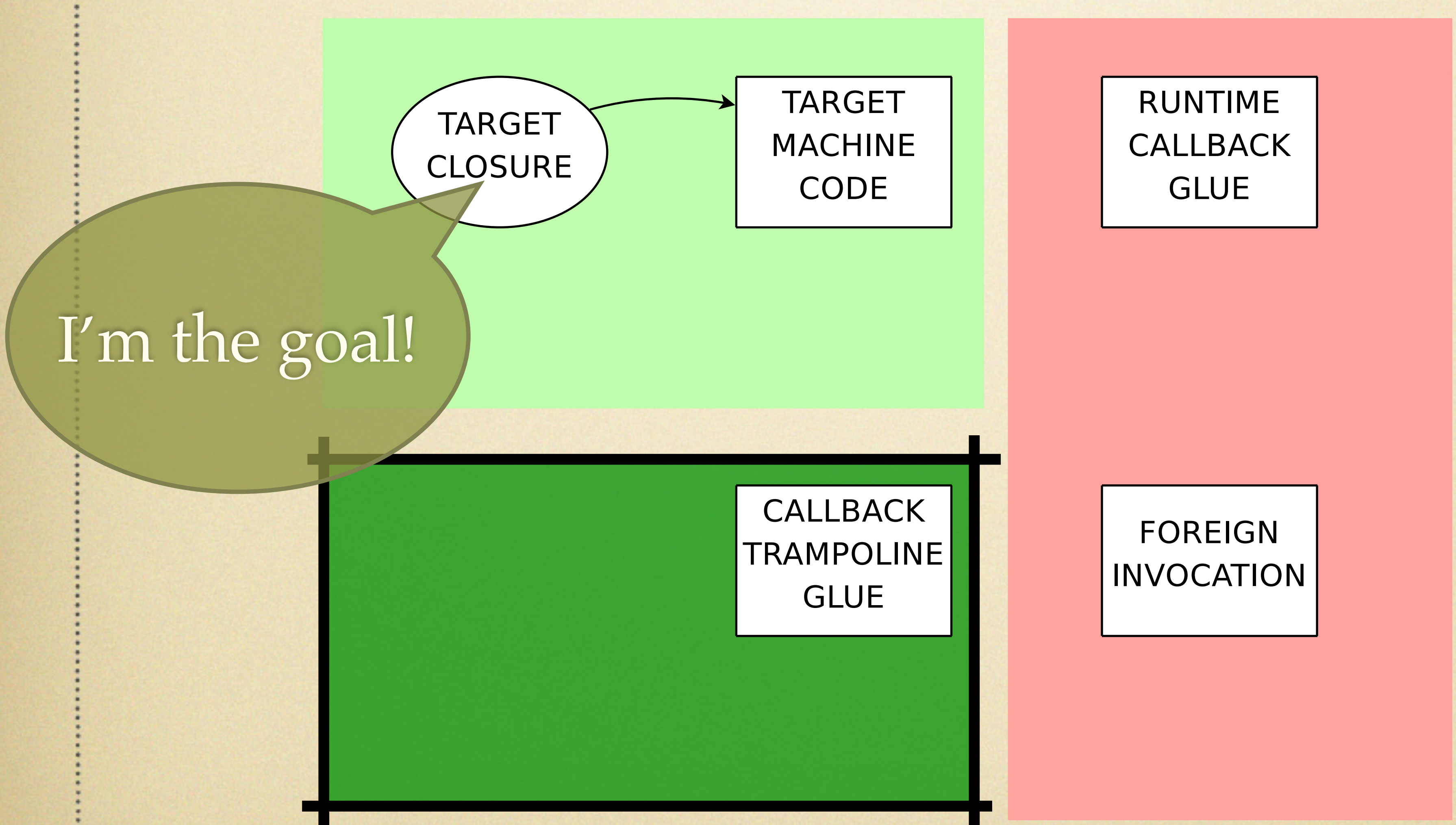
Distributing control amongst heap objects.
Remember: we start at the foreign invocation, go through trampoline, then runtime, and finally hit the target closure (the “Z” in reverse).

Heap Structure: Callbacks



Here, we will start with the target closure, and the goal is to come up with the right glue to make the foreign invocation work from C.
The machine code of the closure is not enough; we need its environment as well.
Somewhere in the glue code we need to get our hands on that closure object.

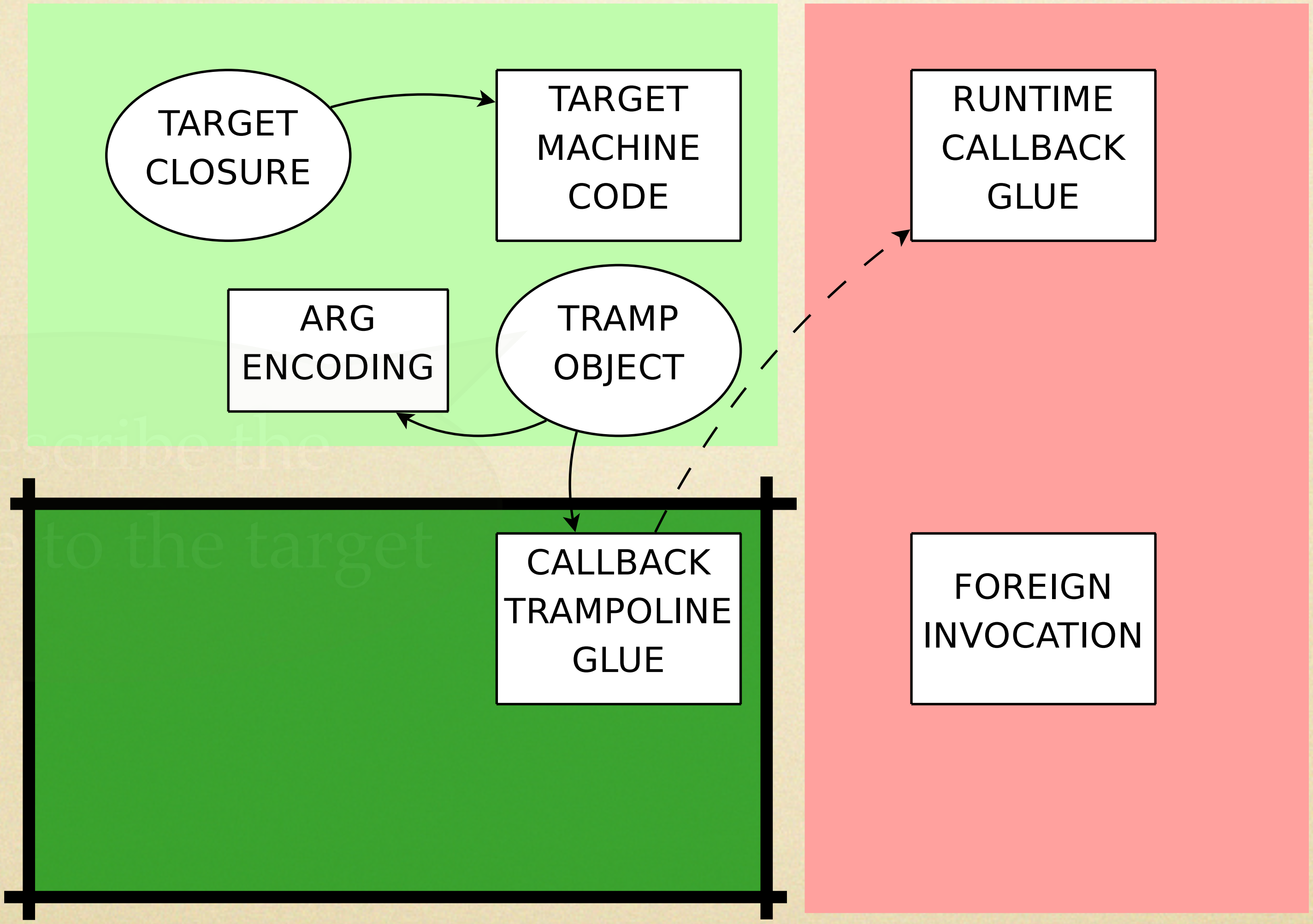
Heap Structure: Callbacks



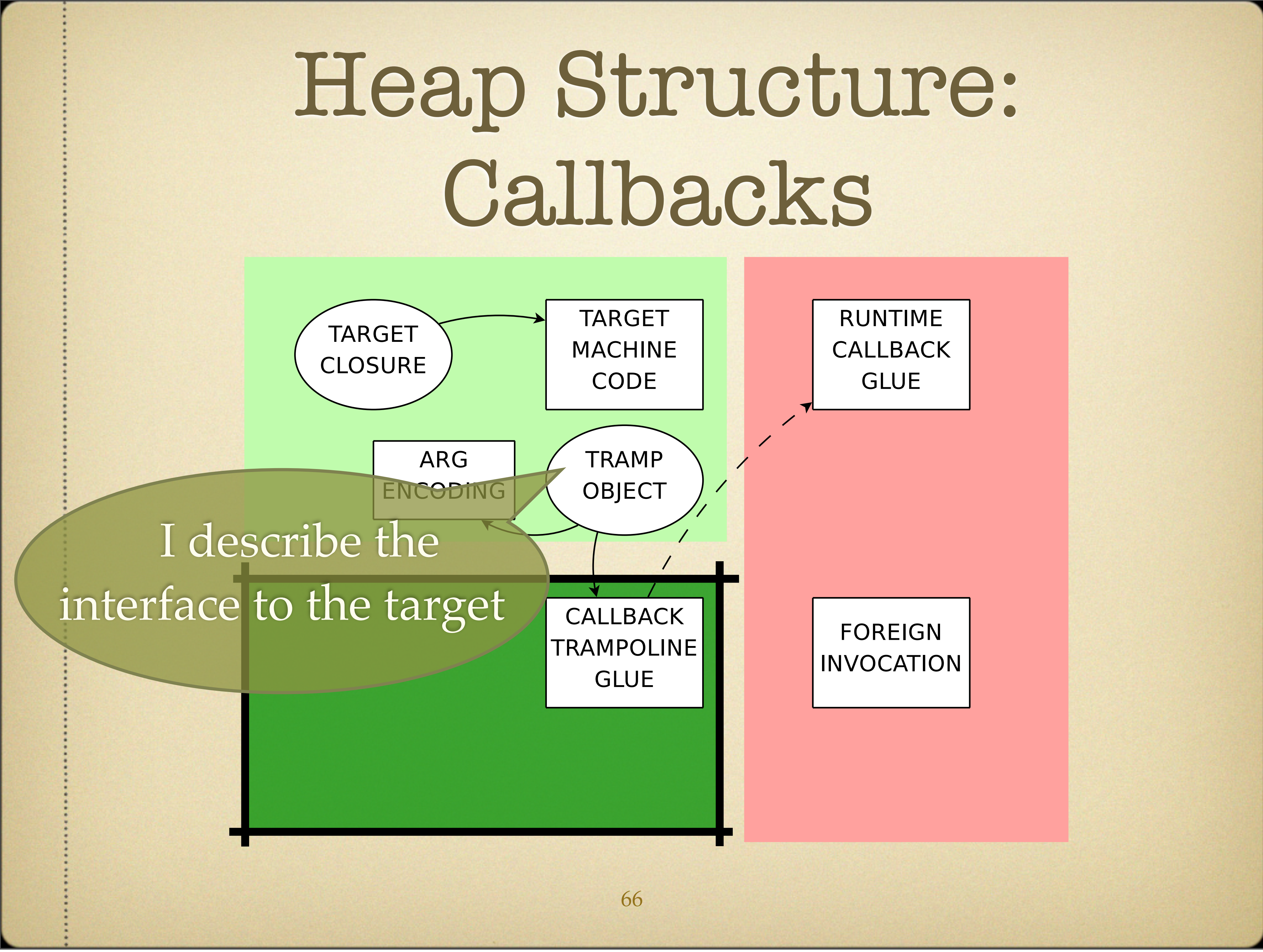
65

Here, we will start with the target closure, and the goal is to come up with the right glue to make the foreign invocation work from C.
The machine code of the closure is not enough; we need its environment as well.
Somewhere in the glue code we need to get our hands on that closure object.

Heap Structure: Callbacks

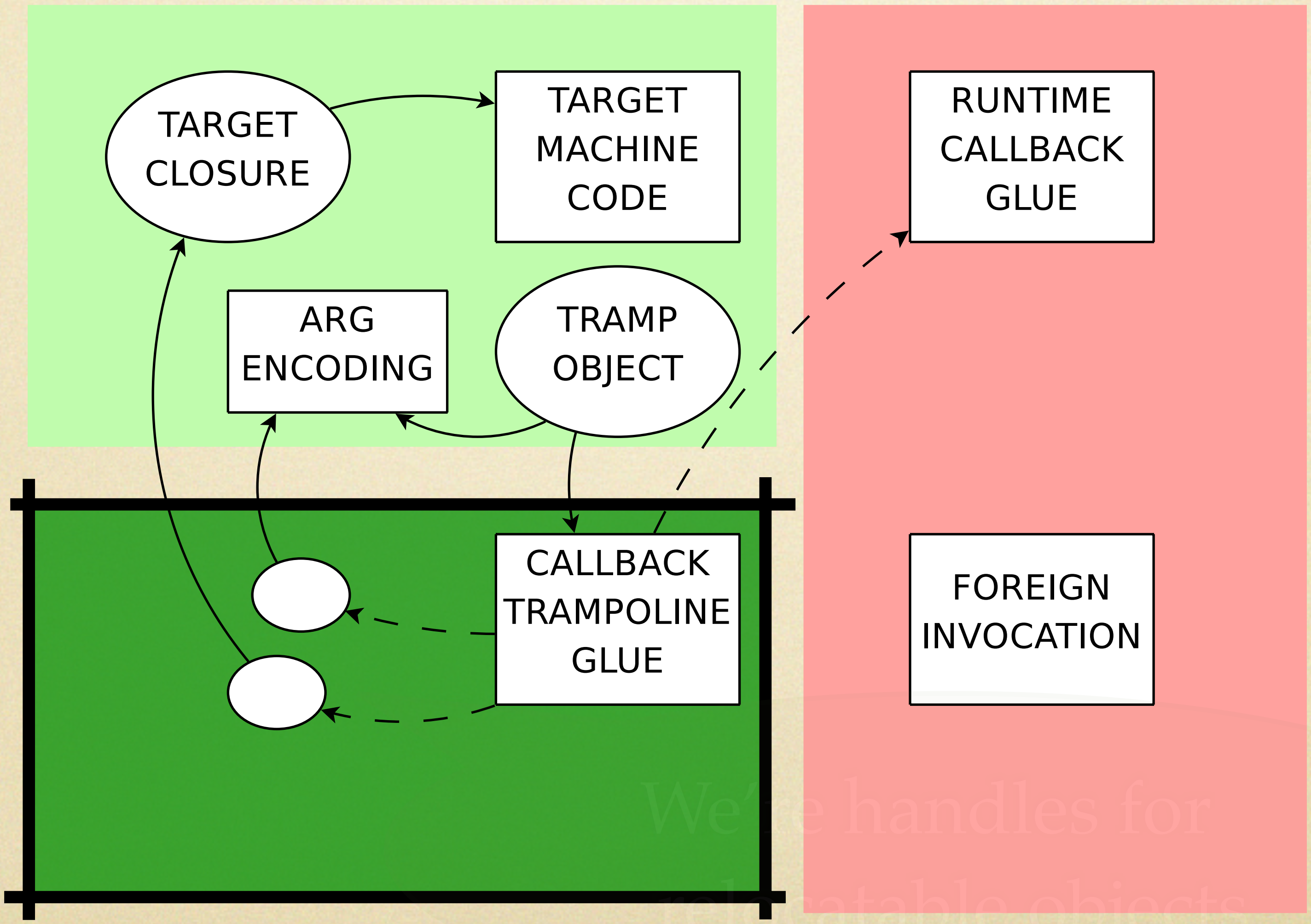


When we construct a callback, we create a trampoline object that creates the machine code as well as a description of the argument encoding.
Both the target closure and the arg encoding need to be passed into the runtime glue code.
Can we put references from the trampoline glue to these objects? (Why not direct? Why not indirect?)
How can we do this without violating our heap structure invariants?



When we construct a callback, we create a trampoline object that creates the machine code as well as a description of the argument encoding.
Both the target closure and the arg encoding need to be passed into the runtime glue code.
Can we put references from the trampoline glue to these objects? (Why not direct? Why not indirect?)
How can we do this without violating our heap structure invariants?

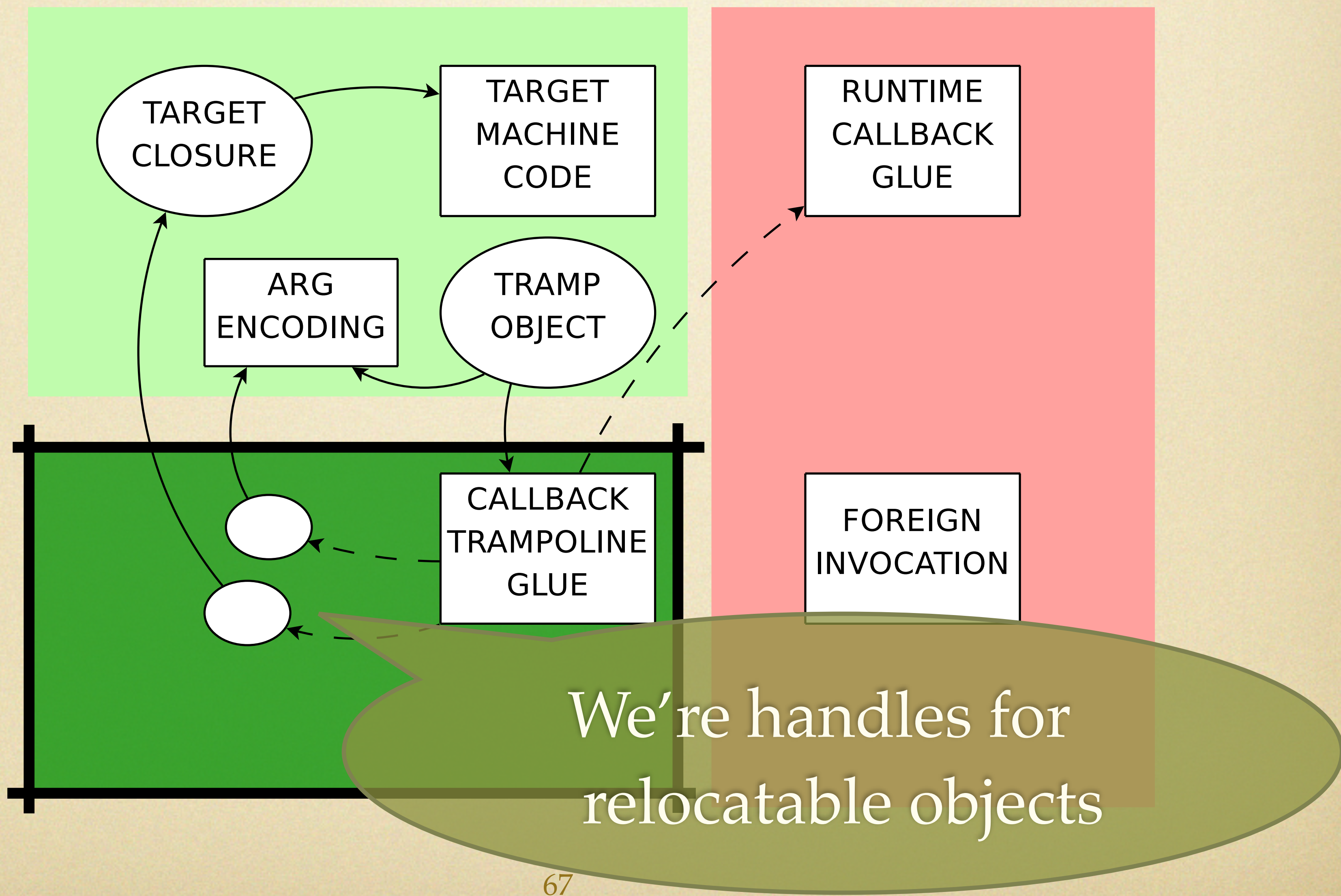
Heap Structure: Callbacks



67

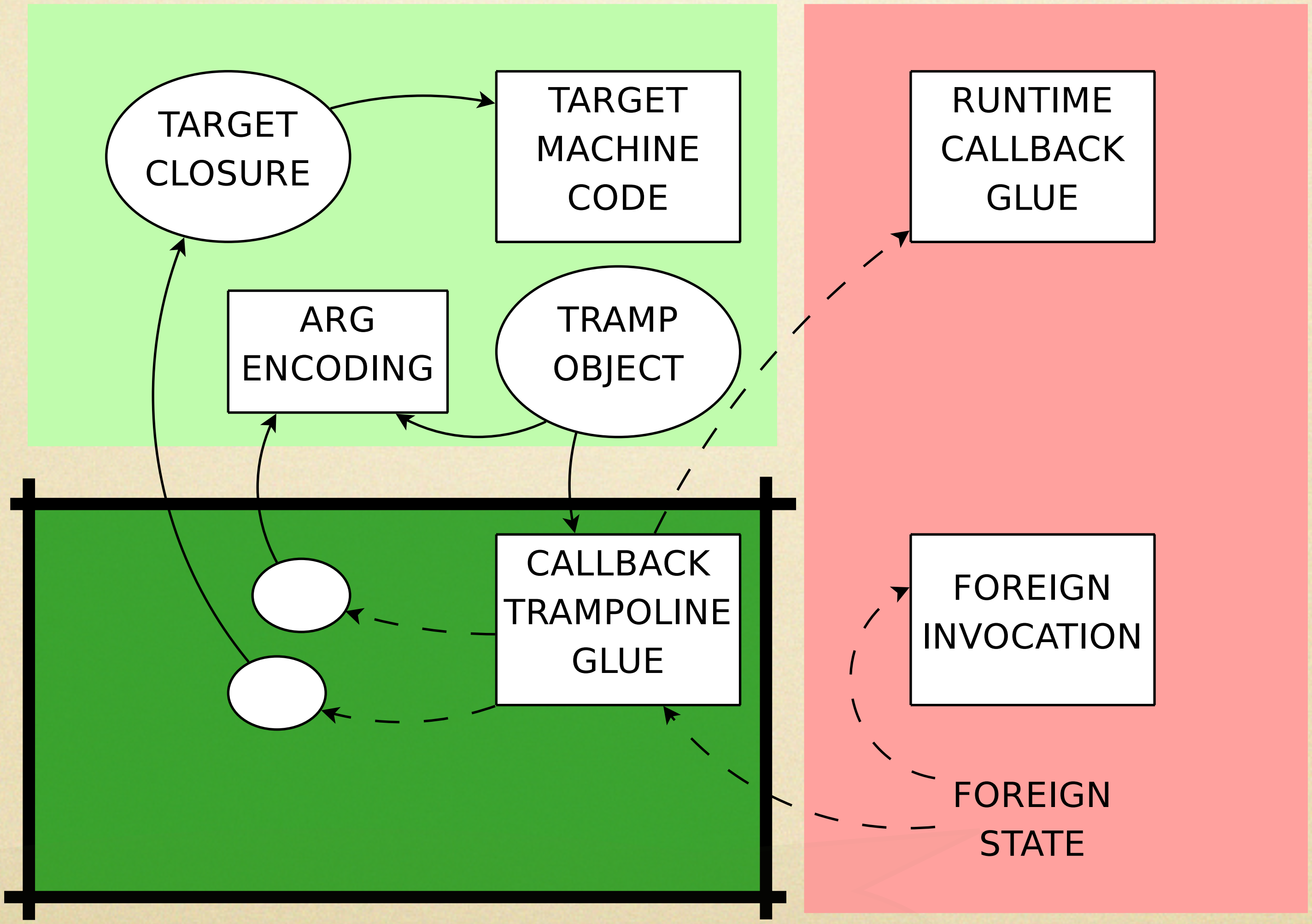
We solve the problem by introducing a level of indirection.
These handles are allocated as non-relocatable, so the trampoline glue can encode indirect references to them, while they themselves can have direct references to the relocatable part of the Scheme heap.

Heap Structure: Callbacks



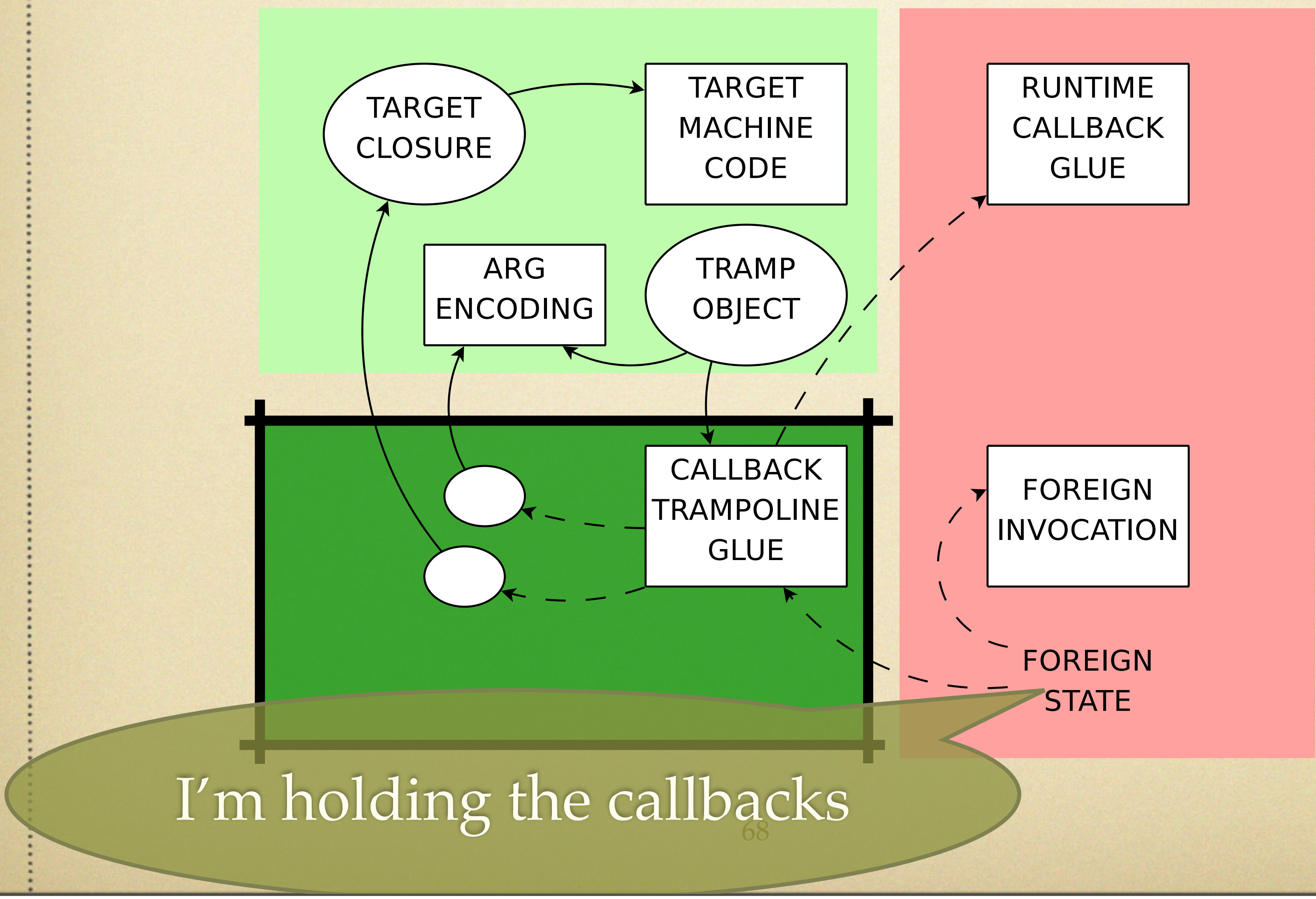
We solve the problem by introducing a level of indirection.
These handles are allocated as non-relocatable, so the trampoline glue can encode indirect references to them, while they themselves can have direct references to the relocatable part of the Scheme heap.

Heap Structure: Callbacks



Finally, to complete the picture, when we pass a callback into the foreign world, the foreign state will have a reference to the trampoline glue (that, as far as C is concerned, is just some C function pointer).
(This is the most complex picture; it is explained in the paper as well.)

Heap Structure: Callbacks



Finally, to complete the picture, when we pass a callback into the foreign world, the foreign state will have a reference to the trampoline glue (that, as far as C is concerned, is just some C function pointer).
(This is the most complex picture; it is explained in the paper as well.)