Implementing the STG

Pedro Vasconcelos

April 10, 2023

Implementing the STG

- Last lecture: abstract machine for STG evaluation
- This lecture: implementing the STG on standard architectures
  - code generation
  - representation of the *heap* and *stacks*

1. Code generation
2. Heap representation
3. Implementation of stacks
4. Transitions of the abstract machine

1. Code generation
2. Heap representation
3. Implementation of stacks
4. Transitions of the abstract machine

Code generation

Option: Generate C code rather than machine code.

Advantages

- facilitates suport for differente architectures
- re-uses optimizations in existing C compilers

Disadvantages

- generating machine code allows specializad otimizations
- standard C does not support everything necessary for a compiler backend (for example: cross function jump to a label)
- may require non-standard extensions (GCC)

Code generation (cont.)

Currently

- GHC suports native code generation for X86, AArch64, PPC and Wasm
- C code generation is no longer enabled by default
- Alternative: generate LLVM intermediate code

  `http://llvm.org/`

Code addresses

Used in three forms:

- to label code blocks;
- kept in data structures (stack, *closures*, . . .);
- as destination for control flow instruction (jump).

How can we represent this in C?

1st Version: use integers

```
#define JUMP(label) ((pc=label),break)

main () {
  int pc = 1;    /* program counter */
  while (TRUE) do
    switch(pc) {
       1:  /* code for label 1 */
           ....
           JUMP(...);
       2:  /* code for label 2 */
       ...
    }
}
```

Disadvantages

- One extra layer of interpretation
- Stresses the C compiler:
  a single function contains all compiled code
- Does not allow for separate compilation

2nd Version: function pointers

- Each basic block is a C function
- The return value is the address of the continuation
- A mini-interpreter executes the blocks in sequence

Return the continuation address

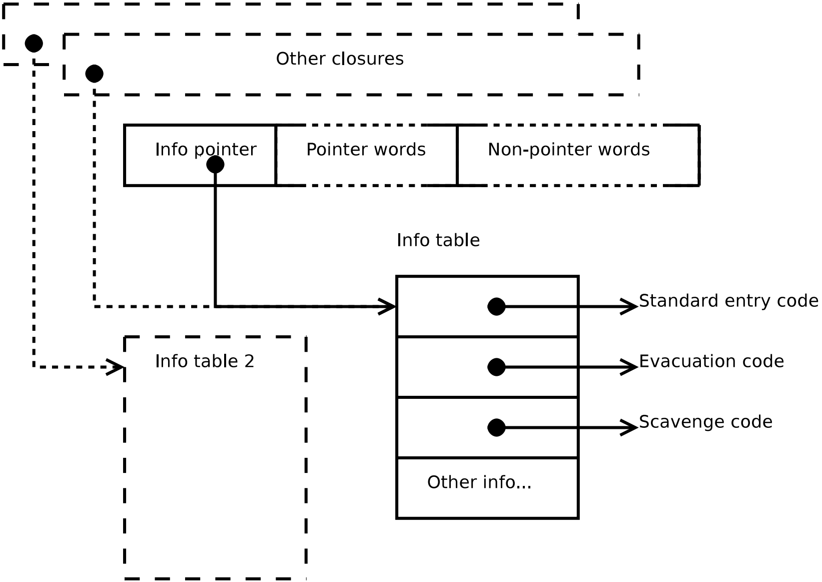
```
typedef void *(*codeptr)(void);
#define JUMP(label) return(label)

codeptr label1() {   /* code for label 1 */
    ....
    JUMP(label2);
}
codeprt label2() {   /* code for label 2 */
    ....
}

int main() {      /* mini-interpreter */
    codeprt pc = label1;
    for(;;) { pc=(*pc)() };
}
```

Advantages

- Code addresses correspond to machine code pointers
- Generates C functions for small code blocks
- Suports separate compilation using the standard *linker*
- Optimizations:
  - avoid creating the stack frame and *frame pointer*
  - generate *assembly* for direct jumps:

```
#ifdef __x86__
/* x86 specific (incomplete!) */
#define JUMP(label)  asm("jmp "#label)
#else
/* portable C */
#define JUMP(label)  return(label)
#endif
```

Representation for *closures*



Representation for *closures*

- An *info* pointer followed by a sequence of words
- The info pointer refers to static information about each kind of closure:
  1. *lambda-forms* declared in the program;
  2. construtors;
  3. indirections
- Pointers for running specific functions:

  standard entry: compiled code for each *lambda-form*;

  evacuation/scavenge: code for garbage collection
- Every closure with of same *lambda-form* shares the static table
- No need for tags or size information

Memory management

Two-space garbage collector:

- Heap divided in two areas: *fromspace* and *tospace*
- Allocations performed in *fromspace* (bumping a pointer)
- When the system runs out of available space:
  1. the colector copies every live closure to *tospace*
  2. the roles of the two areas are swapped

Closure rentention

Closures are live if:

- they are directly accessible from the execution stacks;
- they are accessible from other live closures;

i.e. they form the strongly connected component starting from the roots in the stack.

Requirements for the GC algorithm:

- reproduces the original heap graph;
- handles cycles and sharing;
- executes efficently (in time and space).

Copying GC algorithm

Cheney (1970)

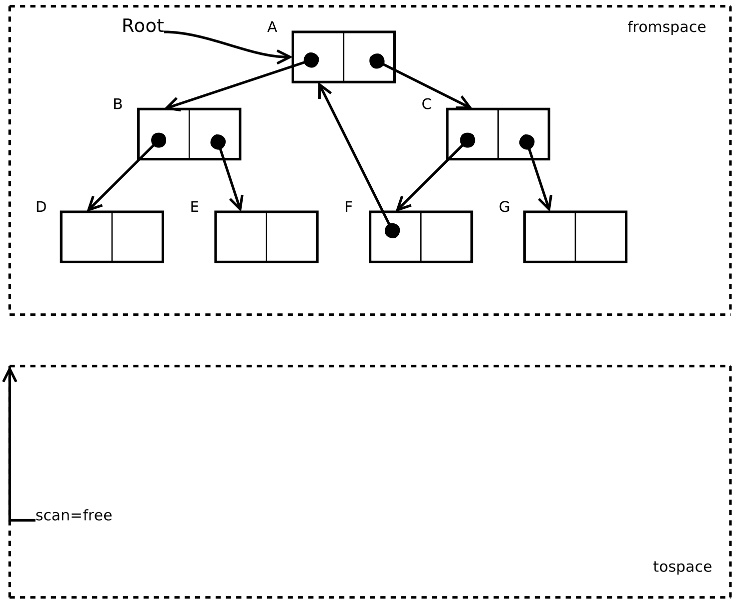
Evacuate: copy a closure *fromspace* $\rightarrow$ *tospace*:

1. places a *fowarding pointer* in the original address.
2. if it has been copied alread: returns immediatly.

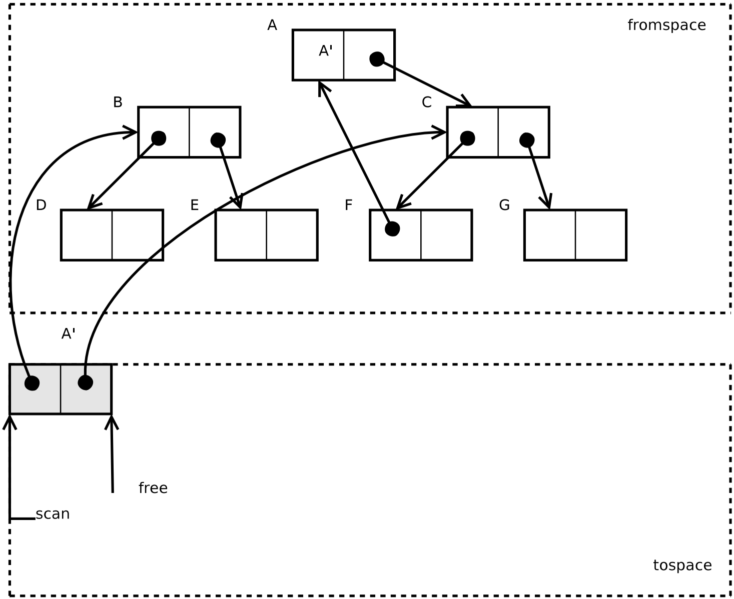
Scavenge: copies children of a closure in *tospace*

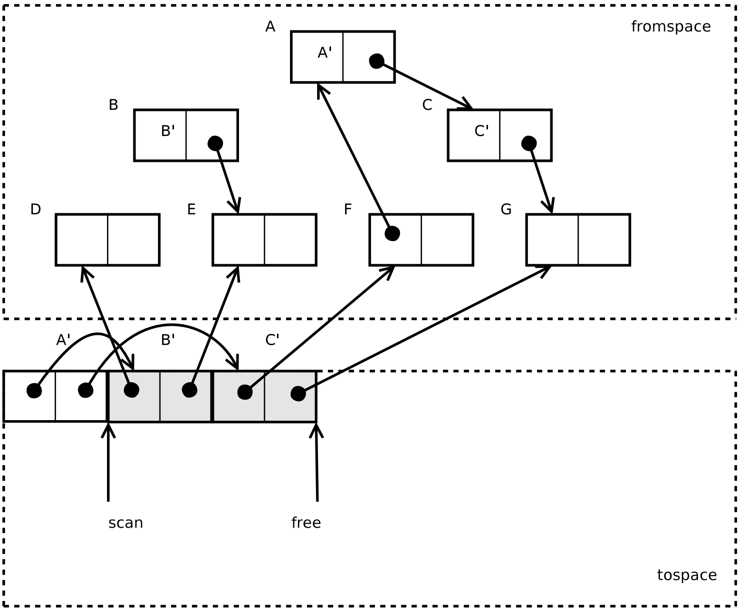
1. calls *evacuate* for each pointer in the copies closure;
2. replaces the pointer by the new address.

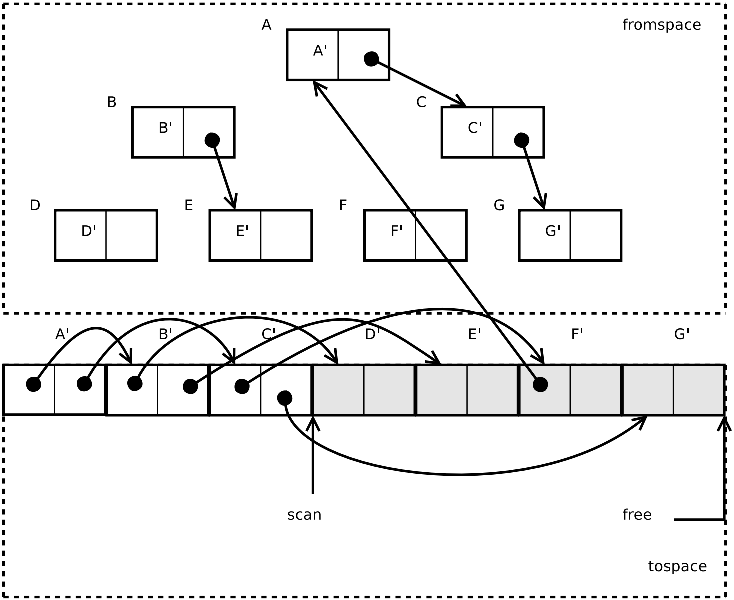
This algorithm is not recursive; instead it implements a queue using two pointers in *tospace*.

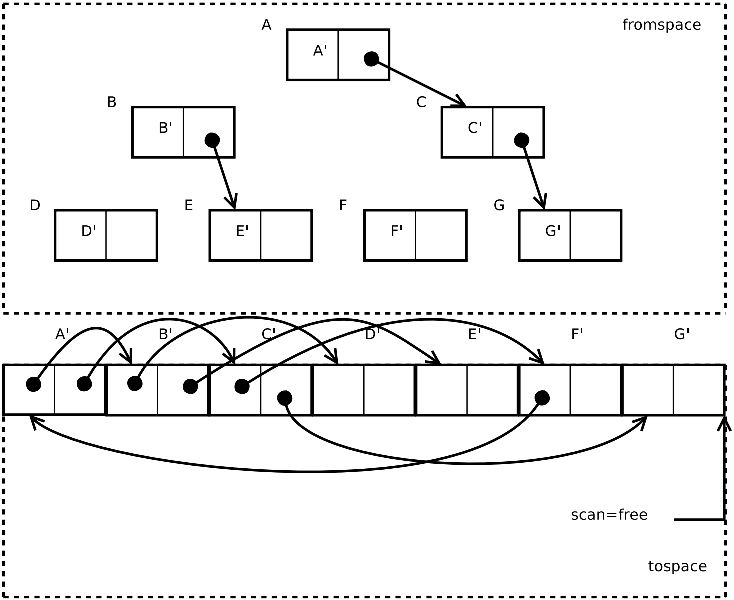


Root
A
fromspace
B
C
D
E
F
G
scan=free
tospace


fromspace
A
A'
B
C
D
E
F
G
A'
free
scan
tospace


fromspace
A
A'
B
B'
C
C'
D
E
F
G
A'
B'
C'
scan
free
tospace


fromspace
A
A'
B
B'
C
C'
D
D'
E
E'
F
F'
G
G'
A'
B'
C'
D'
E'
F'
G'
scan
free
tospace


fromspace
A
A'
B
B'
C
C'
D
D'
E
E'
F
F'
G
G'
A'
B'
C'
D'
E'
F'
G'
scan=free
tospace

Two-space GC: advantages

- Preserves cycles and sharing of data structures
- Compacts the resulting semi-space
- Preserves *locality*: data that is used together is likely to be placed in the same page and cache line
- Facilitates memory allocation: simply bump a pointer in the resulting space
- Executes in constant space and time proportional to the size of live data
- Layout of the closures need to be known only in the *evacuate/scavenge* code
- No need for tags or size information
- Allows treating special cases (e.g. indirections)

Two-space GC: disadvantages

- Uses twice the live memory (2 spaces)
- Variant: *generational collectors*

Implementation of stacks

Three conceptually distinct stacks:

argument stack arguments to functions;

return stack continuations (case alternatives);

update stack update-frames.

How can we implement these?

A single stack?

- The three stacks works in sync
- It should be possible to combine into a single stack
- A contiguous block of words (e.g. *array* of `void *`)

Advantage: less space wasted

Disadvantage: mixing basic values (integers) and pointers in the stack makes GC harder

Two stacks

A-stack stack for pointers

B-stack stack for basic values

- The two stacks grow in oposing directions
- GC only goes throught stack A

Transitions of the abstract machine

- Each basic block is translated into a C function with no arguments sem parâmetros
- Function arguments passed on the stack
- Registers (e.g. stack pointers) kept in global variables

Function aplication

- Push arguments to the stack(s)
- Enter the closures associated to the function
- Global variable `Node` points to the active closure
- Access to free variables by indexes throught `Node`

Example

```
/* apply3 = {} \n {f,x} -> f {x,x,x}; */
/* SpA : top of stack A */

Node = SpA[0];    /* grab closure for f */
t = SpA[1];       /* grab x */
SpA[0] = t;       /* push extra args */
SpA[-1] = t;
SpA = SpA - 1;    /* adjust stack pointer;
  the A-stack grows to lower addresses */
ENTER(Node);       /* enter closure */
```

Enter a closure

- `Node` points to the *closure*
- Executes the code for the *standard entry*
- Always the 1st entry in the *info table*

```
#define ENTER(n)   JUMP((n)->info[0])
```

More information

Not covered:

- *let(rec)* and *case* expressions
- arithmetic operations
- *returns* and *updates*

See the bibliography: *Implementing lazy funcional languages on stock hardware: the Spineless Tagless G-machine*, Simon Peyton Jones, 1992.