Implementação da STG

Pedro Vasconcelos

22 de Maio de 2020

Implementação da STG

- Na aula passada: modelo abstracto da STG
- Nesta aula: implementação da STG numa arquitectura "*standard*":
  - geração de C ou código máquina?
  - representação da *heap* e *stacks*?

1. Geração de código
2. Memória Heap
3. Implementação das pilhas
4. Transições da máquina abstracta

1. Geração de código
2. Memória Heap
3. Implementação das pilhas
4. Transições da máquina abstracta

Geração de código I

Opção histórica: gerar código C em vez de código máquina.

Vantagens

- facilita o suporte de diferentes arquiteturas
- toma partido das optimzações nos compiladores existentes

Geração de código II

Desvantagens

- gerar diretamente código máquina permite otimizações mais específicas
- C *standard* não suporta tudo que é necessário num *backend* de compilador (exemplo: *labels*)
- por vezes é benéfico usar extensões não-portáteis (GCC)

Geração de código III

Atualmente

- GHC suporta geração de código nativo diretamente para X86
- Geração de código C já não é suportada
- Em alternativa: gerar código LLVM

`http://llvm.org/`

Endereços de código

São usados de três formas:

- para etiquetar blocos de código;
- guardados em estruturas de dados (pilha, *closures*, tabelas, etc.);
- usando como destino de uma transferência de controlo.

Como representar em linguagem C?

1ª Versão: usar inteiros

```
#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 */
       ...
    }
}
```

Desvantagens

- Um nível extra de interpretação
- Sobrecarrega o compilador de C:
  uma única função com todo o código compilado
- Não permite compilação separada

2ª Versão: endereços de funções

- Cada bloco básico é uma função em C
- O valor de retorno é o endereço da continuação
- Um mini-interpretador executa os blocos em sequência

Retornar o endereço da continuação

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

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

main() {      /* mini-interpretador */
    codeprt pc = label1;
    while(TRUE) { pc=(*pc)() };
}
```

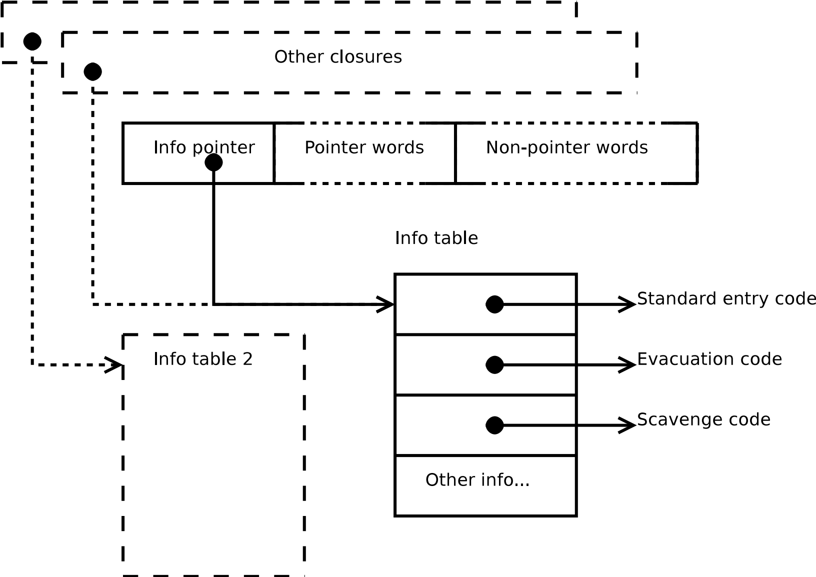
Vantagens

- Endereços de código virtual correspodem directamente a endereços da maquina
- Gera funções em C para blocos de código pequenos
- Suporta compilação separada com o *linker* usual
- Optimizações:
  - evitar salvaguarda de registos ou *frame pointer*
  - gerar *assembly* para efectuar saltos directos:

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

1. Geração de código
2. Memória Heap
3. Implementação das pilhas
4. Transições da máquina abstracta

Representação de *closures*



Representação de *closures*

- Um apontador *info* seguido de uma sequência de palavras
- Informação estática por cada tipo de *closure*:
  1. *lambda-forms* declaradas no programa;
  2. construtores;
  3. indirecções...
- Apontadores para código compilado (funções C):

  standard entry: código para a *lambda-form*;
  evacuation/scavenge: código para recolha de lixo
- *Closures* duma *lambda-form* partilham tabela estática
- Não necessita de etiquetas ou informação de tamanho

Gestão de memória

"*Two-space garbage collector*"

- *Heap* dividida em duas áreas: *fromspace* e *tospace*
- Alocação efectuada no espaço *fromspace*
- Quando se esgota o espaço disponível:
  1. o colector copia todas as *closures* "vivas" para o *tospace*
  2. inverte os papeis das duas áreas

Retenção de *closures*

Devem sobreviver à recolha de lixo as *closures* que:

- são directamente atingíveis da(s) pilha(s) de execução;
- são atingíveis transitivamente por outras *closures*;

i.e. a componente fortemente conexa do grafo com raizes na pilha.

Requesitos:

- reproduzir o grafo original
- tratar correctamente ciclos e partilha
- de forma eficiente (i.e. tempo e espaço)

Algoritmo de cópia

Cheney, 1970

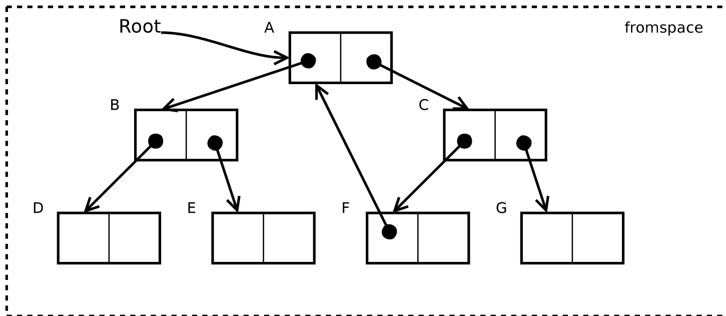
Evacuate: copia uma *closure fromspace* $\rightarrow$ *tospace*:

1. coloca um *fowarding pointer* no endereço original.
2. se já foi copiada: retorna imediatamente.

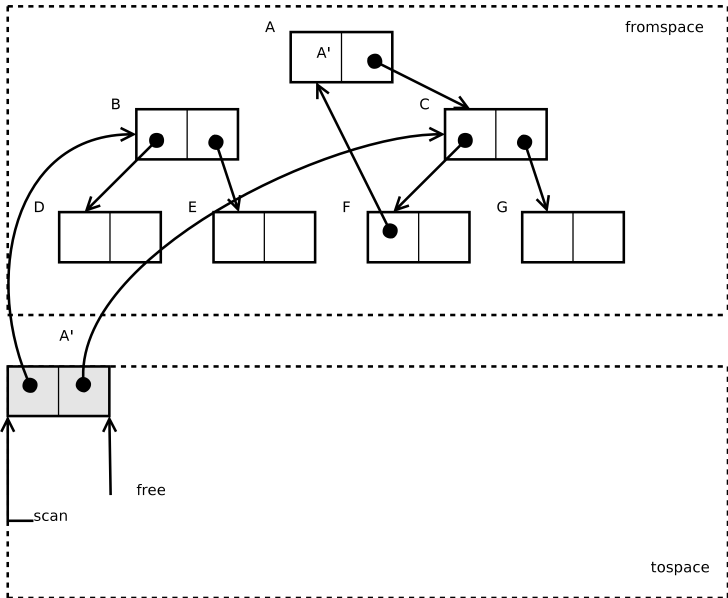
Scavenge: copia descendentes de uma *closure* em *tospace*

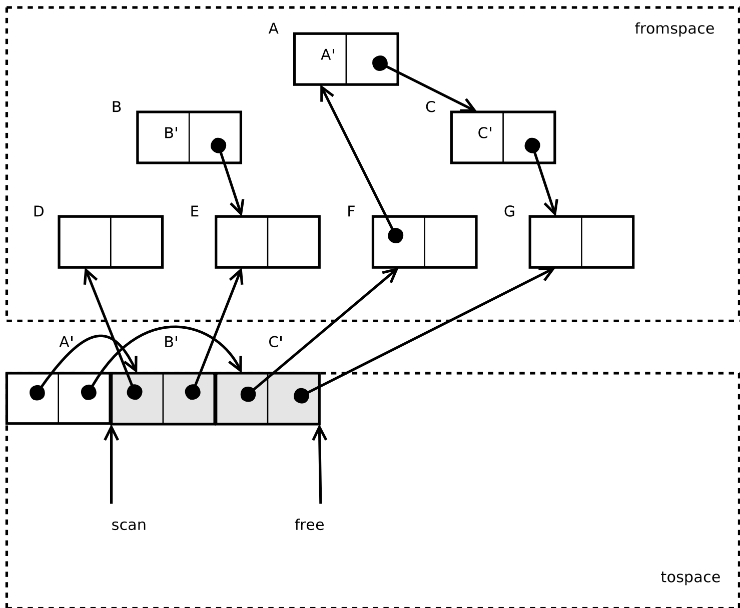
1. invoca *evacuate* para cada apontador na *closure* copiada;
2. substitui o apontador pelo novo endereço.

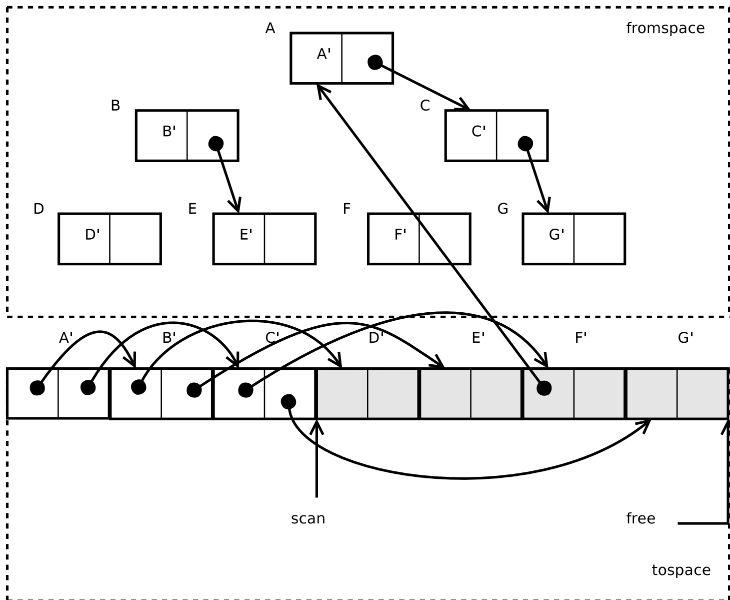
Este algoritmo não é recursivo: implementa uma fila de espera usando dois apontadores em *tospace*.

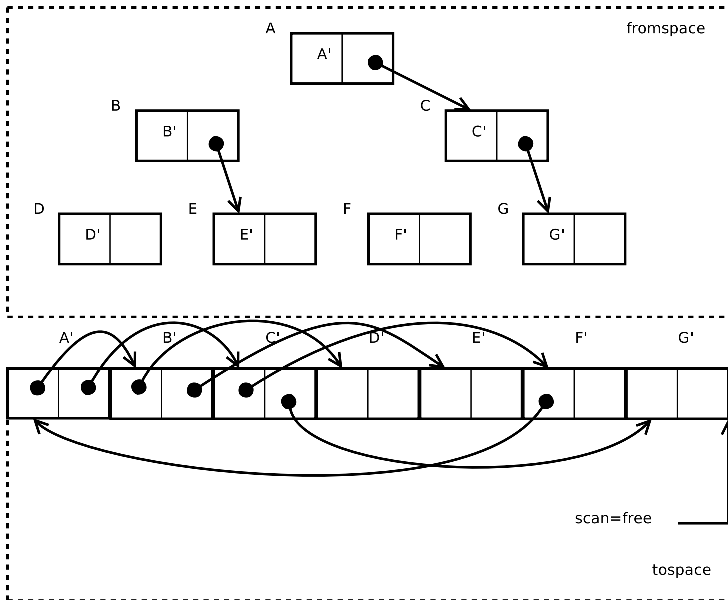

Root
A
fromspace
B
C
D
E
F
G

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: vantagens

- Preserva ciclos e estruturas partilhadas
- Compacta o semi-espaço resultante
- Facilita a alocação de memória
- Executa em espaço constante
- *Layout* das *closures* especificado apenas no código *evacuate/scavenge*
- Não necessita de *tags* ou tamanhos
- Permite tratar casos especiais e.g. indirecções

Two-space GC: desvantagens

- Usa o *dobro* da memoria consumida (dois espaços)
- Variante: *colectores geracionais*

1. Geração de código
2. Memória Heap
3. Implementação das pilhas
4. Transições da máquina abstracta

Implementação das pilhas

Três pilhas conceptualmente distintas:

argument stack argumentos de funções;

return stack continuações (alternativas);

update stack *update-frames*;

Como implementar?

Uma pilha única

- As três pilhas operam em sincronia
- É possível combinar numa só
- Um bloco contínguo de palavras (e.g. *array* de `void *`)

Vantagem: menor desperdício de espaço

Desvantagem: mistura de valores básicos (inteiros) e apontadores dificulta a recolha de lixo

Duas pilhas

A-stack pilha de apontadores

B-stack pilha de valores básicos

- As duas pilhas crescem em sentidos opostos
- Recolha de lixo percorre apenas a pilha A

1 Geração de código

2 Memória Heap

3 Implementação das pilhas

4 Transições da máquina abstracta

Transições da máquina abstracta

- Cada bloco básico traduzido numa função de C sem parâmetros
- Argumentos passados na pilha
- Registos passados em variáveis globais (e.g. endereços de topo da pilha)

Aplicação

- Empurrar argumentos para as pilhas
- Entrar a *closure* associada à função
- Variável global `Node` aponta a *closure* activa
- Acesso às variáveis livres indexado por `Node`

Exemplo

```
/* apply3 = {} \n {f,x} -> f {x,x,x}; */
/* SpA : topo da pilha 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 */
ENTER();           /* enter closure */
```

Entrar numa *closure*

- `Node` aponta para a *closure*
- Executa o código *standard entry*
- É sempre a 1ª entrada da *info table*

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

Mais informação

Falta tratar:

- expressões *let(rec)*, *case*
- operações aritmética
- *returns* e *updates*

Implementing lazy funcional languages on stock hardware: the Spineless Tagless G-machine, Simon Peyton Jones