Universidade de Évora

Compiladores

2ª Frequência - 5 de Maio de 2015

(1,5) 1. O registo de activação contém (a ordem não interessa):

A. As variáveis locais, o program counter e os argumentos da função

B. Os argumentos da função, o return address, o stack pointer antigo e as variáveis locais

C. Um ou mais arrays com as variáveis locais, argumentos da função, stack pointer, frame pointer, return address e return value

D. Os argumentos da função, o frame pointer antigo, o return address e as variáveis locais

(1,5) 2. A geração de código deve ser feita:

A. Antes da análise semântica, quando a APT ainda mostra uma representação "fiel" do programa inicial

B. Depois da análise semântica, porque é aqui que a APT foi transformada numa Symbol Table

C. Depois da análise semântica, pois só nesta fase sabemos que o programa está correcto lexical, sintactica e semanticamente

D. Depois da APT estar ligada correctamente à Symbol Table

3. Considere o seguinte programa em *Ya!*:

```
1  f (n: int) : int {
2    r, i : int = 1;
3
4    while i <= n do {
5      r = r * i;
6      i = i + 1;
7    };
8
9    return r;
10 };
11
12 main () : void {
13   print(f(3));
14 };
15
```

(2) (a) Mostre uma representação da Symbol Table, no final da análise semântica da função `f()`.

Solução: Por exemplo:

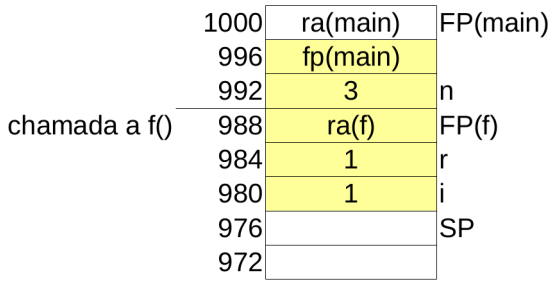
ST:

Nome	Tipo	
f	func: int → int	CTX1
n:arg	int	CTX2
r	int	
i	int	

Como ainda não passámos pela função `main()`, esta ainda não é conhecida, logo não aparece na ST.

(3) (b) Mostre uma representação da stack, durante a execução do programa, imediatamente antes do return da função `f()`. Considere que a stack começa no endereço 1000, cada célula ocupando 32 bits. Coloque anotações, para especificar o que cada célula da stack representa.

Solução: A amarelo temos o registo de activação (frame) inicial de `f()`. A frame de `main()` não tem argumentos nem variáveis locais, logo só tem o backup de `ra` (*return address*). Seria válido haver um `fp` guardado pela "função" que chamou `main()`, mas podemos ignorar essa parte.



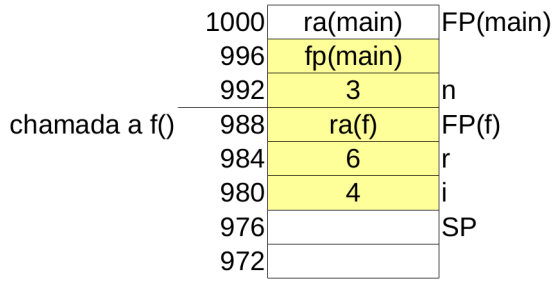
Os valores de `r` e de `i` variam durante a execução de `f()`, mas não mudam de sítio. Temos o `fp` igual a `988`, que nos coloca as variáveis de interesse nas seguintes posições:

- `n` em `fp+4 (992)`;
- `r` em `fp-4 (984)` e
- `i` em `fp-8 (980)`.

(Notar que os argumentos vêm em *offsets* positivos e as variáveis locais em *offsets* negativos)
Simulando a execução de `f()` temos:

1. $r = 1 * 1 (r \leftarrow 1)$
   $i = 1 + 1 (i \leftarrow 2)$
2. $r = 1 * 2 (r \leftarrow 2)$
   $i = 2 + 1 (i \leftarrow 3)$
3. $r = 2 * 3 (r \leftarrow 6)$
   $i = 3 + 1 (i \leftarrow 4)$

Aqui `i = 4`, e a condição do `while` falha, terminando a função. É esta stack que queremos mostrar:



(3) (c) Proponha uma forma eficaz de guardar strings no Registo de Activação, focando-se na independência de plataforma. Ilustre com um exemplo prático.

Solução: Haveria várias formas eficazes (algumas não eficientes) de guardar strings no registo de activação:

1. Assumir que as strings têm um comprimento fixo (p.ex. 100 bytes) e reservar esse espaço na frame.
2. Analisar o código (através da APT) e descobrir o comprimento máximo que cada string atinge no programa. Assim é possível colocar um comprimento máximo em cada variável do tipo string, e reservar apenas esse espaço no RA.
3. Não colocar as strings na frame, mas sim um apontador (endereço) para onde está a string:
   (a) Na heap (usando o equivalente ao `malloc()` do C;
   (b) Na área .GLOBAL, reservando previamente espaço necessário para albergar todas as strings do programa.

(4) (d) Usando a abordagem de "máquina de pilha" sugerida nas aulas, mostre uma possível representação da função `f()`, em assembly MIPS.

(2) (e) Explique o motivo da inclusão do *Return Address* no Registo de Activação. Ilustre com um exemplo dos problemas que podem ocorrer se o *Return Address* não for guardado.

Solução: O return address é guardado logo no início da função. Primeiro, há arquitecturas que não dispõem de um registo `$ra`, sendo a instrução `call` *quem* coloca o return address na stack.

Por outro lado, mesmo havendo um registo `$ra`, assim que é chamada outra função, o valor desse registo é substituído, perdendo-se o anterior... É por isto que temos **sempre** de guardar o return address na frame.

Um exemplo simples é uma função `f()` que no seu corpo chama uma função `g()`: quando `g()` é chamada (`jal g`), perde-se o valor de `$ra` que tinha sido colocado pelo `jal f`. Quando é feito o return de `g()` (`jr $ra`), o salto é feito para a próxima instrução de `f()`, como era esperado. No entanto, quando se fizer o return de `f()`, vamos saltar outra vez para a mesma instrução (a que segue a chamada a `g()`).

Por isso, guardamos sempre (e repomos, antes do *return*) o return address.

(3) (f) Suponha que pretende implementar estruturas (`structs`) na linguagem *Ya!*. Para este efeito, considere o formato "normal" das `structs` da linguagem C. Discuta as implicações desta nova componente da linguagem na análise semântica.

Solução: