Compiladores

Análise de Longevidade

Introdução

Nas Notas de Aula sobre Geração de Código Intermediário & Otimização, formas de gerar representações intermediárias e de gerar otimizações no código durante a compilação foram apresentadas. Partindo desta base, a proposta deste material é abordar o intervalo de vida das variáveis, importante para realizar e otimizar a alocação de registradores.

O objetivo destas notas de aula é fazer com que o leitor compreenda os conceitos relacionados à análise de longevidade e como esta técnica se relaciona com as etapas de otimização e alocação de registradores.

Análise de Longevidade

O código abaixo, retirado de Dubach (2021), modifica os valores das variáveis `x` e `y` com base em operações envolvendo seus endereços e valores armazenados:

```
.data
x: .space 4
y: .space 4
.text
la $t0, x
lw $t1, ($t0)
add $t2, $t0, $t1
la $t3, y
```

```
lw $t4, ($t3)
sub $t5, $t4, $t2
add $t6, $t2, $t4
sw $t5, ($t0)
sw $t6, ($t3)
```

Porém, o código faz uso de sete registradores temporários, de `$t0` a `$t6`. O que fazer quando há menos do que sete registradores disponíveis?

A ***Análise de Longevidade*** (***Liveness Analysis***) tem como objetivo determinar o período de tempo em que uma variável é necessária em um programa, ou seja, o período entre sua criação e destruição. Este período de tempo é chamado de ***longevidade da variável*** ou ***intervalo de vida*** (***live range***).

Diz-se que uma variável está viva quando ela está em uso e ainda pode ser utilizada no futuro (RIGO, 2009).

A definição de Dubach (2021) diz que uma variável também pode ser entendida como um registrador virtual, e ela está viva em algum ponto do programa se estiver previamente ***definida*** por alguma instrução e será ***usada*** por uma instrução no futuro. Caso contrário, diz-se que ela está morta.

Nesse sentido, variáveis diferentes (`a` e `b`) podem "compartilhar" o mesmo registrador se elas nunca forem usadas e/ou definidas ao mesmo tempo, ou seja, se elas nunca estiverem vivas ao mesmo tempo.

Portanto, o estudo da Análise de Longevidade é importante para

- alocação otimizada de memória, principalmente registradores;
- evitar vazamentos de memória[1];
- evitar problemas de execução como a sobreposição indesejada de dados ou perda de

[1] Vazamento de memória (*memory leak*) é um fenômeno que ocorre quando um programa gerencia mal as alocações de memória, não liberando-a após o uso.

dados.

Alguns compiladores utilizam a Análise de Longevidade como parte de seu processo de otimização, o que pode resultar em programas mais rápidos.

A Análise de Longevidade utiliza a informação do fluxo de controle do programa e das referências a variáveis para determinar o tempo de vida de cada variável. Com base nesse tempo de vida, é possível alocar e liberar a memória de maneira eficiente. Portanto, o alocador de registradores usa as informações de longevidade.

Conforme Jones (2019), geralmente a longevidade das variáveis é analisada da perspectiva de uma instrução. Cada instrução, por vezes modelada como um vértice em um Grafo de Fluxo de Controle (explicado na sequência) tem um conjunto associado de variáveis vivas.

Exemplo:

```
(...)
31: int r = s + t;
32: return u * v;
```

Assumindo que 31 e 32 são os números das linhas em um código válido, representa-se por *live(31)* o conjunto de variáveis vivas na linha 31. No exemplo, *live(31) = {s, t, u, v}*.

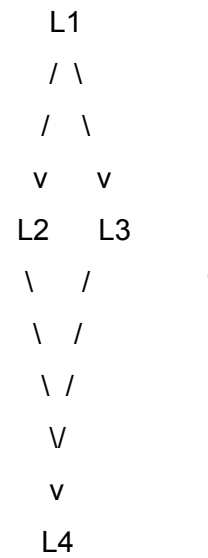
Control Flow Graph – Grafo de Fluxo de Controle

A análise do código para verificar a longevidade das variáveis nem sempre é uma tarefa simples, pois o código pode conter estruturas condicionais e de repetição, sendo necessário compreender o seu fluxo dinâmico (DUBACH, 2021).

Uma das formas de analisar o fluxo de controle de um programa é a partir da modelagem do

código em um grafo, conhecido como ***Grafo de Fluxo de Controle*** (**Control Flow Graph**[2] – CFG). Um CFG é uma forma gráfica de representação para o fluxo de controle de um programa através de um grafo direcionado. Cada nó (vértice) de um CFG representa uma instrução ou um conjunto de instruções e cada aresta direcionada do grafo representa o fluxo de controle para o código fornecido.

L1 → L2 é equivalente a ter a sequência de código L1; L2.



Significa que há uma instrução L1, seguida por L2 ou L3. Depois, tanto L2 quanto L3 levariam a L4. Portanto, os caminhos possíveis são:

1. L1; L2; L4
2. L1; L3; L4.

Exercício: modele o código abaixo, adaptado de Rigo (2009), em um Grafo de Fluxo de Controle:

```
      a := 1
L¹:   b := a + 1
      c := c + b
    a := b * 2
    if a < 10 goto L¹
    return c
```

[2] Criado e publicado por Frances Elizabeth Allen, em 1970, no artigo intitulado *Control Flow Analysis*. Obs.: Frances foi a primeira mulher a receber o *Turing Award*, em 2006, fruto de diversas contribuições em teoria e prática de otimização de compiladores. Ela faleceu em 2020.

Resposta: disponível na Figura 1, abaixo.

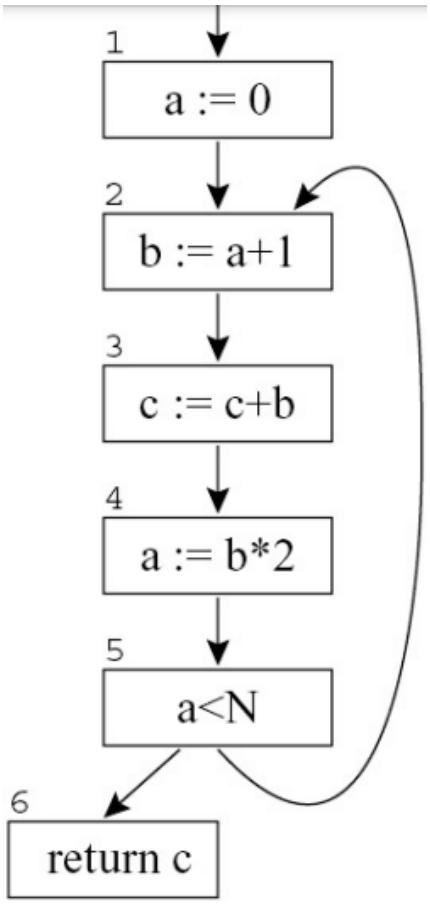


Figura 1. Resposta do exercício. Fonte: Rigo, 2009.

Quantos registradores são necessários neste código? A resposta pode ser obtida observando o Grafo de Fluxo de Controle e gerando o *live range* de cada variável.

A partir deste código e observando o grafo, nota-se que a variável `b` está viva nas arestas 2→3 e 3→4. Portanto, o *live range* de `b` é {2→3, 3→4}.

A variável `a` está viva nas arestas 1→2, 4→5 e 5→2. Portanto, o *live range* de `a` é {1→2, 4→5, 5→2}.

A variável `c` está viva em todo programa, portanto o *live range* de `c` é {1→2, 2→3, 3→4, 4→5, 5→2, 5→6}.

Mas por que `c` está viva em todo programa?

Nota-se que `c` foi definida (recebeu a expressão através da operação de atribuição) e usada (na expressão a ser atribuída, `c + b`) na linha 3. No entanto, para ser usada, ela já precisava ter um valor. O contexto é desconhecido, mas é possível assumir aqui que o valor foi inicializado antes. Por exemplo, pode ser que `c` seja um parâmetro para a função que contém as instruções do exemplo. Desta forma, assume-se que `c` já estava viva no início do programa, pois, do contrário, ocorreria uma falha (um *bug*). Este caso será resgatado na sequência.

As variáveis `a` e `b` nunca estão vivas simultaneamente, então é possível afirmar que elas podem compartilhar um registrador! Assim, dois registradores são suficientes para este programa.

É importante observar que o *live range* de uma variável pode ser contínuo ou distribuído em diferentes partes do código. Isso significa que uma variável pode estar viva em um instante, morta no instante seguinte e viva novamente em outro momento.

Como vimos, é importante analisar as arestas para obter os intervalos de vida de uma variável. Toda aresta é direcionada e, assim, tem um nó (vértice) antecessor (origem) e um nó (vértice) sucessor (destino).

- Chama-se de `pred[n]` o conjunto de todos os antecessores do nó *n*.
- Chama-se de `succ[n]` o conjunto de todos os sucessores do nó *n*.

Retornando ao exemplo da Figura 1, todas as arestas que possuem um nó (vértice) *n* como entrada são ditas *out-edges*. As arestas que possuem este nó como saída são ditas i*n-edges*. Agora, analisando o quinto vértice na Figura 1, percebe-se que ele possui uma aresta de entrada (4→5) e duas arestas de saída (5→2 e 5→6). Portanto, as arestas *out-edges* do vértice 5 são 5→6 e 5→2. Já a única aresta *in-edge* do vértice 5 é 4→5.

Ainda sobre a Figura 1, pode-se dizer que
pred[5] = {4}
succ[5] = {2, 6}

Quem é pred[2]? pred[2] = {1, 5}
Quem é succ[2]? succ[2] = {3}

Existem ainda os conjuntos *use* e *def*:

- use[n] é o conjunto de variáveis usadas em *n*, onde *n* é um nó (vértice) no CFG e *v* é uma variável.
- use[v]: é o conjunto de nós (vértices) no CFG onde *v* é usado.
- def[n] é o conjunto de variáveis definido por *n*, onde *n* é um nó (vértice) no CFG e *v* é uma variável.
- def[v]: é o conjunto de nós (vértices) no CFG onde *v* é definido.

No exemplo, o nó 3 possui `c := c + b`. Então
use[3] = {c, b}
def[3] = {c}

Quem é def[c] = ?

Conforme Dubach (2021), uma definição mais precisa de longevidade diz que uma variável *v* está viva em uma aresta do CFG se

- Existe um caminho direcionado da aresta para o uso de *v* (o vértice $\in$ use[b]); e
- O caminho não possui uma definição de *v* (o vértice $\notin$ def[v]).

A Figura 2 traz o slide de Dubach (2021) com a definição de longevidade, e a Figura 3 traz a análise da longevidade do CFG do exemplo da Figura 1.

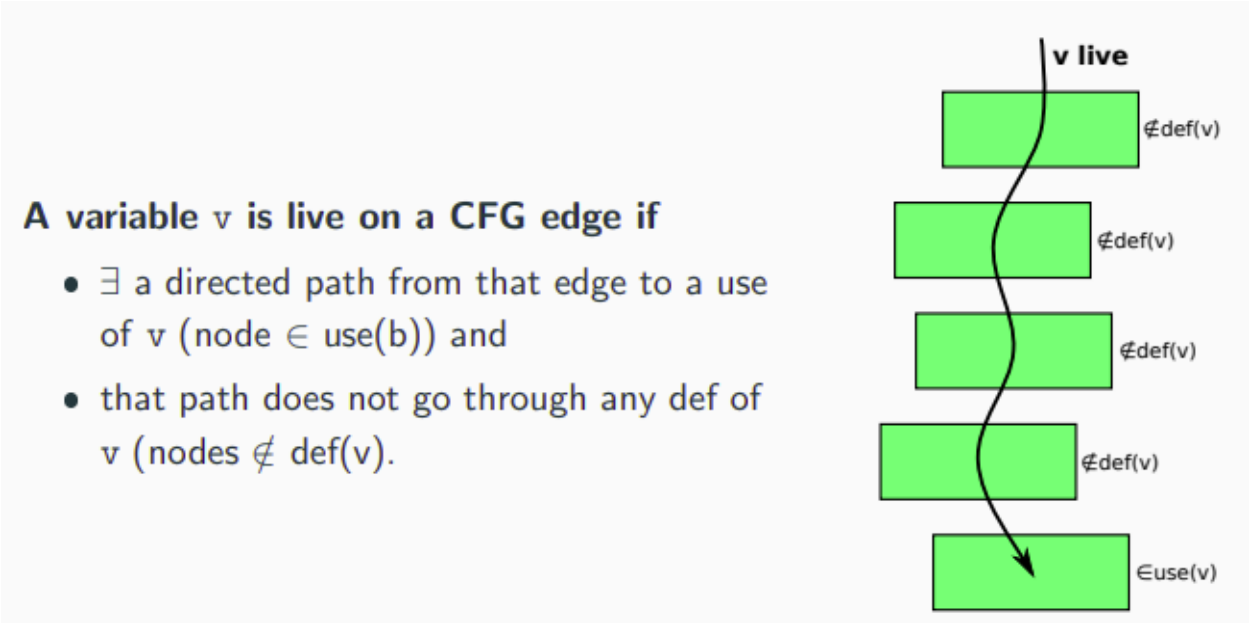


Figura 2. Longevidade de uma variável *v*. Fonte: Dubach, 2021.

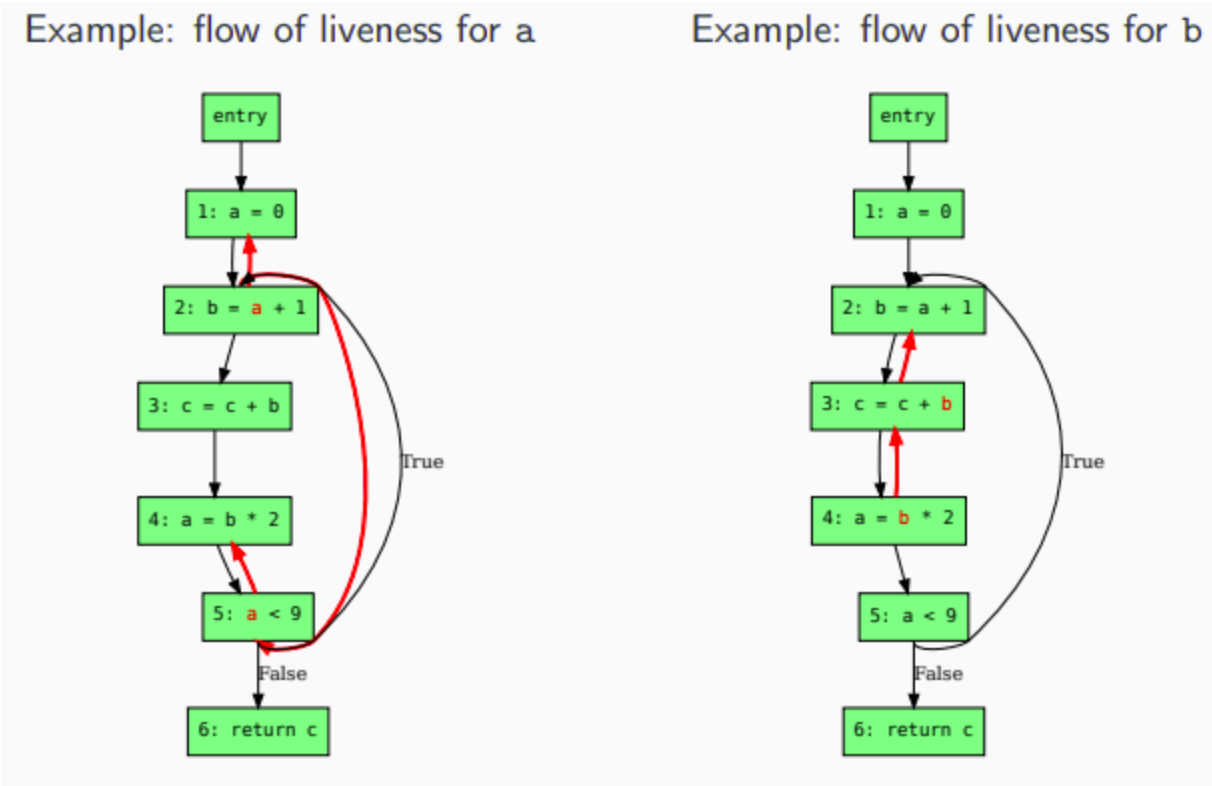


Figura 3. Fluxo de longevidade de `a` e `b`, usados no exemplo de aula. Fonte: Dubach, 2021.

Nota-se na Figura 3 que a análise é feita no sentido inverso. Observa-se onde a variável *v* é

utilizada e percorre-se o grafo para trás até que o vértice pertença a def[v], pois o comportamento de nós futuros é quem determina a longevidade em um determinado nó.

Sabendo disso, sugere-se realizar como exercício a identificação do conjunto use[c] e o fluxo de longevidade desta variável.

Conclusão

A Análise de Longevidade é uma técnica que ajuda a garantir que programas sejam executados de maneira eficiente e sem erros. Alocar e liberar a memória de maneira correta é fundamental para garantir a *performance* de um programa e a Análise de Longevidade é uma ferramenta valiosa para alcançar esse objetivo.

Exercícios

1) A análise de longevidade pode ajudar a otimizar um código? Se sim, como?
2) O que são blocos básicos e como eles podem auxiliar no uso dos CFG?
3) Considere o seguinte programa para responder às duas questões abaixo:

```
1: int fun(int n) {
2:     int a = n + 5;
3:     int b = a * a;
4:     int c = n + b;
5:     return c - 1;
6: }
```

   a) Crie um Grafo de Fluxo de Controle para o código acima.
   b) Responda "V" para as afirmações verdadeiras e "F" para as falsas. Justifique as respostas.
      ( ) `a` está viva em {1->2}.
      ( ) `n` está viva em {1->2}.
      ( ) `c` está viva em {4->5}.
      ( ) `x` está viva em {3->4}.

Referências

DUBACH, Christophe. Lecture 16: Liveness Analysis. 2021. Disponível em: https://www.cs.mcgill.ca/~cs520/2021/slides/16-liveness.pdf. Acesso em: 02 jun. 2024.

JONES, Timothy M. Lecture 3: Live variable analysis. Disponível em: https://www.cl.cam.ac.uk/teaching/1819/OptComp/slides/lecture03.pdf. Acesso em: 02 jun. 2024.

RIGO, Sandro. Análise de Longevidade. 2009. Disponível em: https://www.ic.unicamp.br/~sandro/cursos/mc910/2009/slides/cap10-liveness.pdf. Acesso em: 02 jun. 2024.