Compiladores
(IF688)

Leopoldo Teixeira
lmt@cin.ufpe.br | @leopoldomt

Níveis de abstração

- A escolha de um nível de abstração ajuda a determinar quais operações o compilador pode manipular e otimizar
- Por exemplo, uma AST *near-source* torna fácil encontrar todas as referências a um *array* X
- Ao mesmo tempo, esconde todos os detalhes para cálculo de endereçamento necessário para acessar um elemento de X
- Ao usar código de três endereços, o cálculo é exposto, ao custo de obscurecer o fato de que se refere a X

Mapeando valores em nomes

- Similarmente, a disciplina que um compilador usa para atribuir nomes internos aos valores computados durante execução pode afetar o código que pode ser gerado
- O esquema de nomes pode expor oportunidades de otimização ou obscurecê-las

Nomeando valores temporários

- A representação intermediária geralmente tem mais detalhes do que o código fonte
- Alguns detalhes são implícitos no código fonte, outros resultam de escolhas deliberadas na tradução

Static Single Assignment

- Disciplina de nomes que muitos compiladores modernos usam para codificar informação sobre o fluxo de controle e dados de valores
- Na forma SSA, nomes correspondem unicamente a pontos específicos de definição no código
- Cada nome é definido apenas uma vez
- Como um corolário, cada uso de um nome como argumentos de operação carrega informação sobre o ponto onde o valor foi originado

Static Single Assignment

- Um programa está na forma SSA, quando:
  - (1) cada definição tem um nome distinto;
  - (2) todo uso se refere a uma única definição
- Para transformar um programa em SSA, é necessário inserir funções especiais, denominadas phi, em pontos onde o fluxo de controle converge

Exemplo

Original code

```
x = ...
y = ...
while (x < 100) {
  x = x+1
  y = y+x
}
```

Exemplo

Original code

```
x = ...
y = ...
while (x < 100) {
   x = x+1
   y = y+x
}
```

Code in SSA form

x_0 = ...
y_0 = ...
if (x_0 >= 100) goto L0
L1: $x_1 = \phi(x_0, x_2)$
$y_1 = \phi(y_0, y_2)$
$x_2 = x_1 + 1$
$y_2 = y_1 + x_2$
if (x_2 < 100) goto L1
L0: $x_3 = \phi(x_0, x_2)$
$y_3 = \phi(y_0, y_2)$

Static Single Assignment

- A forma SSA foi intencionada para otimização de código
- A propriedade de single-assignment permite ao compilador ficar alheio a muitas questões associadas ao tempo de vida de valores
    - nomes nunca são redefinidos ou mortos, o valor está sempre disponível a partir de um caminho

Introdução a otimizações de código

Quando realizar otimizações

- AST?
- Linguagem de máquina?
- Representação intermediária?
- Quais são os prós e contras?

Quando realizar otimizações

- **AST**
  - pró: independente de máquina
  - contra: muito alto nível, poucas oportunidades

Quando realizar otimizações

- **Linguagem de máquina**
  - pró: aproveita oportunidades de otimização de hardware
  - contra: dependente de máquina, precisa ser reimplementada ao realizar *retargeting*

Quando realizar otimizações

- **Representação intermediária**
  - pró: independente de máquina, expôe mais oportunidades de otimização
  - contra: mais uma linguagem para se preocupar

Otimização

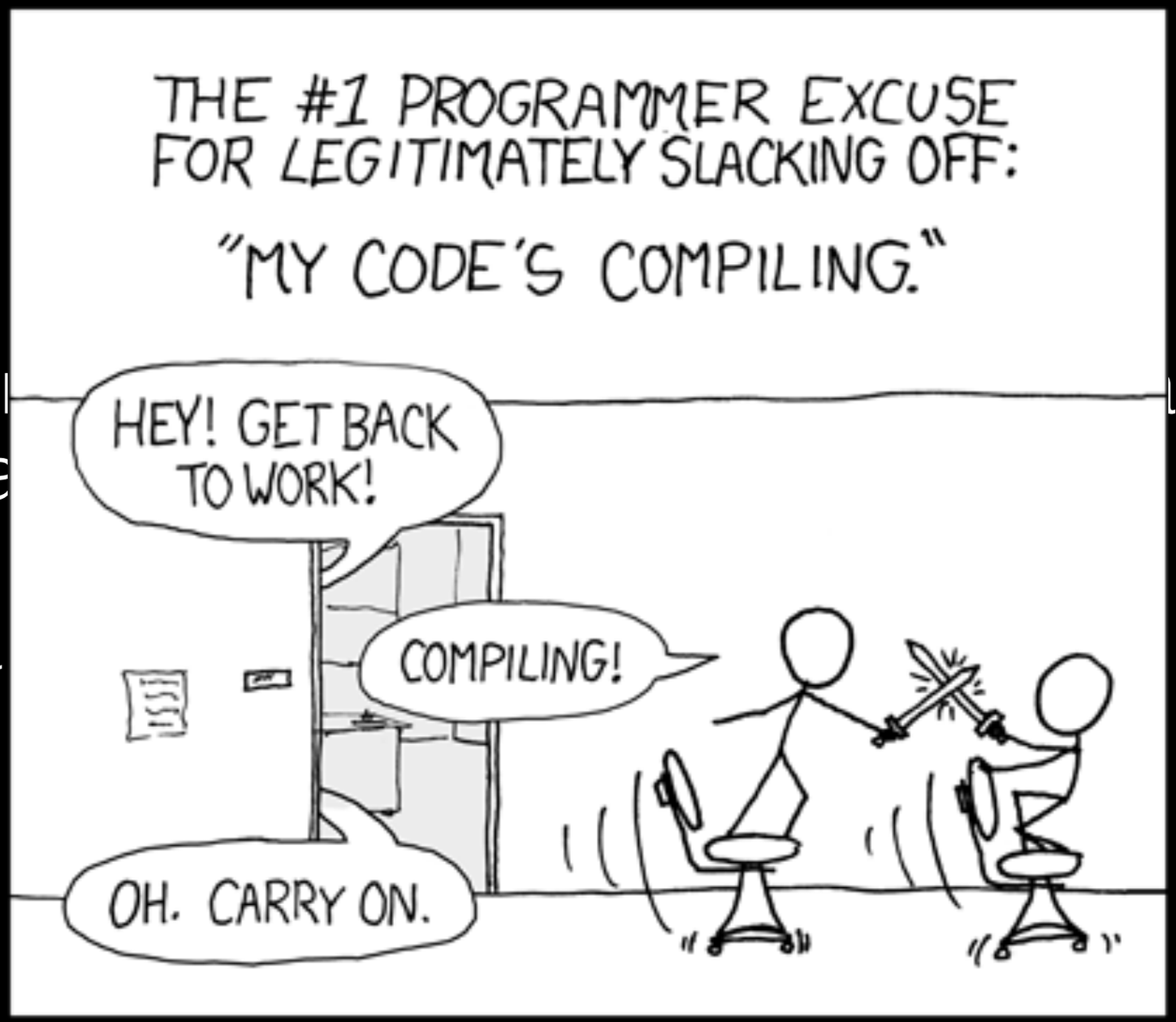
- Visa melhorar a utilização de algum recurso
  - tempo de execução (em geral)
  - tamanho de código
  - envio de mensagens de rede
  - energia
- Não deve alterar comportamento
  - programa deve executar da mesma forma

Custo de Otimizar

- Em geral, não se costuma decidir aplicar a otimização mais ‘fancy’ conhecida.

- Qual a razão?

Custo de Otimizar



- Em ge… …r a otimiza…
- Qual a…

https://xkcd.com/303/

Custo de Otimizar

- Em geral, não se costuma decidir aplicar a otimização mais ‘fancy’ conhecida.
- Qual a razão?
  - complexidade de implementação
  - certas otimizações são muito caras de se implementar, em tempo de execução
  - as melhores são caras e difíceis
- Objetivo: máxima melhoria com menor esforço

Descobrir, em tempo de compilação, informação sobre o comportamento em tempo de execução do programa

Usar esta informação para melhorar o código gerado pelo compilador

Considerações...

- Como já dito, aplicar a transformação não deve alterar o sentido do programa
- Só aplicar transformações com segurança (*safety*) e benefício (*profitability*)
- Se algum destes não for verdade, não vale a pena aplicar a transformação

Safety

- Corretude é o critério mais importante que o compilador deve satisfazer
- Como saber que uma transformação é segura?
- O que é o sentido de um programa?

Profitability

- Qual a vantagem de aplicar uma transformação como *loop unrolling*?
  - diminuir quantidade de iterações
  - evitar trabalho duplicado
  - *memory bound*

Granularidade

- Local
  - aplicada a blocos básicos isoladamente
- Global (intra-procedural)
  - aplicada a um CFG isoladamente
- Inter-procedural
  - aplicada entre fronteiras de métodos

Otimizações Locais

- Forma mais simples
- Não é necessário analisar o corpo completo do procedimento/método
- Apenas o bloco básico em questão
- Exemplos: simplificações algébricas, *constant folding*

Simplificação Algébrica

- Certas instruções podem ser removidas
  - `x = x + 0`
  - `x = x * 1`
- Algumas instruções podem ser simplificadas
  - `x = x*0  ->`
  - `x = x**2  ->`
  - `x = x*8  ->`
  - `x = x*15  ->`

Simplificação Algébrica

- Certas instruções podem ser removidas
  - `x = x + 0`
  - `x = x * 1`
- Algumas instruções podem ser simplificadas
  - `x = x*0  ->  x = 0`
  - `x = x**2  ->  x = x*x`
  - `x = x*8  ->  x = x<<3`
  - `x = x*15  ->  t = x<<4; x = t-x`

Constant Folding

- Operações em constantes podem ser calculadas antes da execução do código
- Em geral, se há uma instrução do tipo **`x = y op z`** e **`y`** e **`z`** são constantes, podemos calcular **`y op z`**
- Exemplo
  - **`x = 2+2`** **`–> x = 4`**
  - **`if 2<0 jump L`** **`–> pode ser removida`**

Otimizações de Fluxo

- Eliminar código inalcançável
  - código inalcançável no CFG
    - remover blocos básicos que não são alvo de desvios ou sequências de blocos
- Pode tornar o programa menor
  - (potencialmente) mais rápido

Single Assignment form

- Representação que visa facilitar otimizações de código

- Muitas otimizações podem ser simplificadas se cada atribuição é feita a um temporário que ainda não apareceu no bloco básico

- Código de três endereços pode ser rescrito na forma de *static single assignment* (SSA)

SSA

- Reescrevendo
  - `x = a + y  ->  x = a + y`
  - `a = x  ->  a1 = x`
  - `x = a * x  ->  x1 = a1 * x`
  - `b = x + a  ->  b = x1 + a1`

Eliminando *common subexpressions*

- Assumindo que o bloco básico está em SSA
- Então todas as atribuições com o mesmo lado direito computam o mesmo valor
- **`x = y + z  ->  x = y + z`**
- ...
- **`w = y + z  ->  w = x`**

Copy propagation

- Se **`w = x`** aparece em um bloco, todos os usos de **`w`** podem ser substituídos por usos de **`x`**
- Exemplo
  - **`b = z + y  ->  b = z + y`**
  - **`a = b      ->  a = b`**
  - **`x = 2 * a  ->  x = 2 * b`**
- Isto não reduz o tamanho do programa, mas... ?

Copy propagation + Constant folding

- Exemplo
  - `a = 5  -> a = 5`
  - `x = 2 * a  -> x = ?`
  - `y = x + 6  -> y = ?`
  - `t = x * y  -> t = ?`

Dead Code Elimination

- Se **`w = exp`** aparece em um bloco, e **`w`** não é mais utilizado em nenhum ponto do programa
- Então, a instrução é *dead code* e pode ser eliminada
- *Dead code* == não contribui ao resultado do programa
  - **`x = z + y -> b = z + y -> b = z + y`**
  - **`a = x -> a = b -> x = 2 * b`**
  - **`x = 2 * a -> x = 2 * b`**

Aplicando otimizações locais

- Cada otimização tem efeito pequeno
- Tipicamente otimizações interagem
  - realizar uma otimização habilita outras
- Compiladores que realizam otimizações geralmente as aplicam até que não haja mais melhoria possível
- Interpretadores e JIT devem ser rápidos
  - portanto pode ser necessário parar otimização

Exemplo

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Simplificação Algébrica

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

Simplificação Algébrica

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

Copy Propagation

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

Copy Propagation

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

Constant Folding

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

Constant Folding

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

Common Subexpression Elimination

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

Common Subexpression Elimination

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

Copy Propagation

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

Copy Propagation

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

Dead Code Elimination

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

Dead Code Elimination

```
a := x * x
```

```
f := a + a
g := 6 * f
```

Como implementar?

- Assuma que queremos eliminar expressões redundantes de um bloco básico
- Uma expressão ***e*** é redundante em ***p*** se já foi avaliada em todos os caminhos que levam a ***p***

Exemplo

```
a := b + c
b := a - d
c := b + c
d := a - d
```