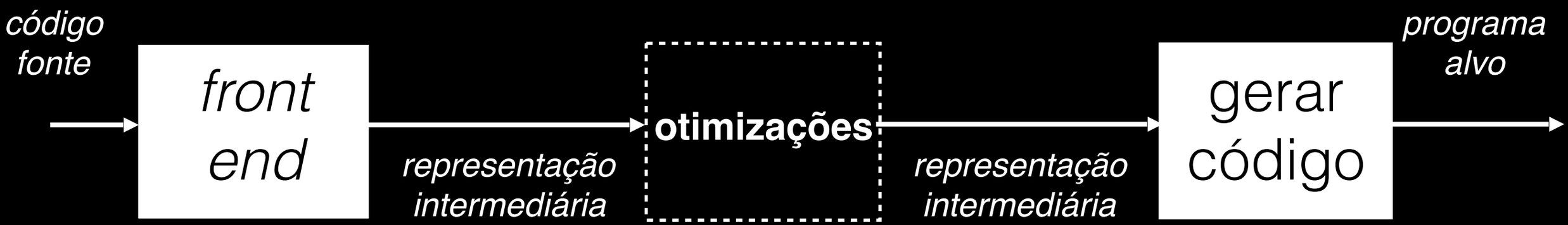
Compiladores
(IF688)

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

Geração de Código



Requisitos

- Código gerado deve preservar a semântica do programa original
- Fazer uso efetivo dos recursos da máquina alvo
- Rodar eficientemente

Gerar programa ótimo é indecidível

Na prática, nos contentamos com heurísticas que geram código bom, mas não, ótimo

Geração de Código

- Seleção de instruções
- Alocação de registradores
- Ordenação de instruções

Questões de Projeto

- Tarefas listadas estão presentes no projeto de praticamente todo gerador de código
  - Embora detalhes sejam especificamente dependentes da representação intermediária, linguagem alvo, ambiente de execução, etc.
- O critério mais importante é a corretude

Entrada

- A entrada consiste da representação intermediária produzida no *front-end*
  - código de três endereços, *bytecode*, árvores sintáticas, DAGs, notação pós-fixa
- Assumimos que a IR já passou pela análise semântica, type-checking, etc.
  - a geração de código assume que a entrada está livre de tais erros

Programa Alvo

- A arquitetura da máquina alvo tem impacto direto na construção de um gerador de código que produza resultados de alta qualidade
- Arquiteturas mais comuns são RISC, CISC e *stack-based*
- RISC: instruções de três endereços, endereçamento simples, conjunto de instruções reduzido, mais registradores
- CISC: menos registradores, instruções de dois endereços, variedade de endereçamentos, classes de registradores, instruções com *side effects*, etc.

Stack-based

- Insere operandos na pilha e realiza operações no topo da pilha, normalmente mantido em registradores

- Reviveram com a JVM, interpretador de *bytecode* Java

- Para superar overhead de interpretação, utiliza-se compiladores *just-in-time* (JIT)

  - traduzem *bytecode*, em tempo de execução, para código de máquina

Nesta aula, utilizaremos um modelo RISC simples como máquina alvo.

Seleção de Instruções

- O gerador de código deve mapear o programa em IR para uma sequência executável na máquina alvo
- A complexidade deste mapeamento é determinada por fatores como
  - nível da IR
  - natureza da arquitetura de instruções
  - qualidade desejada do código gerado

Nível da IR

- Se a IR é de alto nível, o gerador pode traduzir instruções usando templates de código

- No entanto, este tipo de geração pode produzir código que ainda precisa ser otimizado

- Se a IR reflete detalhes de baixo nível, próximos da máquina, o gerador pode utilizar esta informação para gerar código mais eficiente

Instruction-set

- A natureza do conjunto de instruções da máquina alvo também tem forte efeito na dificuldade da seleção de instruções
- Por exemplo, uniformidade: se a máquina não suporta cada tipo de dados de maneira uniforme, pode ser necessário adicionar muitos tratadores para casos especiais
  - em algumas máquinas, operações de ponto flutuante são feitas em registradores separados

```
x = y + z
```

Tradução:

```
LD   R0, y          //R0 = y
ADD  R0, R0, z      //R0 = R0 + z
ST   x, R0          //x = R0
```

```
a = b + c
d = a + e
```

```
x = y + z
```

Tradução:

```
LD   R0, y
ADD  R0, R0, z
ST   x, R0
```

```
x = y + z
```

Tradução:

```
LD   R0, y
ADD  R0, R0, z
ST   x, R0
```

```
a = b + c
d = a + e
```

```
LD   R0, b
ADD  R0, R0, c
ST   a, R0
LD   R0, a
ADD  R0, R0, e
ST   d, R0
```

```
x = x + 1
```

Tradução:

```
LD   R0, x
ADD  R0, R0, #1
ST   x, R0
```

```
x = x + 1
```

Tradução:

```
LD   R0, x
ADD  R0, R0, #1
ST   x, R0
```

```
x = x + 1
```

Tradução:

```
INC x
```

Alocação de Registradores

- Problema chave em geração de código
  - Que valores colocar em quais registradores?
  - Unidade computacional mais rápida da máquina, mas em quantidade limitada
  - Instruções envolvendo registradores são geralmente mais curtas e rápidas das que envolvem operandos na memória
    - daí a importância de utilização eficiente

Uso de Registradores

- Geralmente subdividido em dois problemas
  - Alocação de registradores, onde escolhemos o conjunto de variáveis que residirá em registradores a cada ponto do programa
  - Atribuição de registradores, onde escolhemos qual registrador especificamente será utilizado
- Encontrar uma atribuição ótima é um problema NP-completo, mesmo em máquinas simples

Convenções

- A geração pode ser complicada por certas convenções no uso de registradores
- Certas máquinas requerem pares de registradores para alguns operandos e resultados
- Por exemplo, em algumas máquinas a multiplicação e divisão de inteiros envolve pares de registradores

Convenções

- Instrução de multiplicação: **`M x, y`**
  - **`x`** é o registrador ímpar de um par (*even/odd*) de registradores
  - **`y`** pode estar em qualquer lugar
  - o produto ocupa o par de registradores (*even/odd*)

Convenções

- Instrução de divisão: **`D x, y`**
  - **`x`** é o registrador par
  - **`y`** pode estar em qualquer lugar
  - após a divisão, o registrador par (*even*) guarda o resto e o registrador ímpar (*odd*) o quociente

Ordem de Avaliação

- Ordem de execução das computações pode afetar eficiência do código alvo
- Escolher melhor ordem no caso geral é um problema difícil
- Existem algoritmos para agendamento de instruções em máquinas *pipelined*, que permitem a execução de várias operações em um único ciclo do *clock*

Linguagem Alvo

- Familiaridade com a máquina alvo e o seu conjunto de instruções é importante no projeto de um gerador de código
- Utilizamos uma linguagem alvo similar à assembly, para um computador simples, por ser representativo de várias máquinas

Modelo simples de máquina

- Nossa máquina alvo utiliza instruções de três endereços que permitem operações para carregar e armazenar valores, desvios, e computações sobre valores
- Computador é uma máquina endereçável com *n* registradores de propósito geral (R_0, R_1,… R_{n-1})
- Utilizaremos um conjunto limitado de instruções, assumindo que todos os operandos são inteiros

Load operations

- **`LD dst, addr`**
- Carrega o valor no endereço **`addr`** para localização **`dst`**
- Esta instrução denota a atribuição **`dst`** = **`addr`**
- A forma mais comum é **`LD r, x`**, onde carregamos o valor em **`x`** no registrador **`r`**
- **`LD` r_1`,` r_2** copia valores entre registradores

Store operations

- **`ST x, r`**

- Guarda o valor no registrador **`r`** na posição de memória **`x`**

- Esta instrução denota a atribuição **`x`** = **`r`**

Computation operations

- **`OP dst, src`$_1$`, src`$_2$**
- O efeito desta instrução é a aplicação da operação **`OP`** aos valores em **`src`$_1$** e **`src`$_2$**, armazenando o resultado em **`dst`**
- **`OP`** é um operador como **`ADD`** ou **`SUB`**, e os demais são posições de memória, não necessariamente distintas
- A instrução **`SUB r`$_1$`, r`$_2$`, r`$_3$** denota **`r`$_1$** = **`r`$_2$ `- r`$_3$**
  - qualquer valor em **`r`$_1$** é perdido

Jump operations

- **`BR L`**
  - Transfere o controle para a instrução de máquina marcada por **`L`**. (**`BR`** == *branch*)
- **`B`*`cond`* `r, L`**
  - desvio condicional, onde **`r`** é um registrador, **`L`** é um *label*, e ***`cond`*** é um teste sobre o valor em **`r`**
  - Por exemplo: **`BLTZ r, L`** causa um desvio se o valor armazenado em **`r`** for menor que zero

Endereçamentos

- Assumimos que a máquina alvo tem uma variedade de modos de endereçamentos da memória, para *locations* (posições de memória)
- Uma *location* pode ser um nome de variável **`x`**, referenciando a localização reservada na memória para **`x`** (isto é, o *l-value* de **`x`**)

Indexando

- Podemos também referenciar endereços de forma indexada: **`a(r)`**, denotando a posição computada a partir do *l-value* de **`a`** somado ao valor armazenado no registrador **`r`**.

- **`LD` R_1`, a(`R_2`)`**
  R_1 `=` *contents*`(a +` *contents*`(`R_2`))`

  - ***contents*`(x)`** denota o conteúdo do registrador ou posição de memória representado por **`x`**

- Modo de endereçamento útil para acessar *arrays*

Indexando por inteiro

- Podemos também referenciar endereços de forma indexada por inteiros e registradores
- **LD R_1, 100(R_2)**

  $R_1 = contents(100 + contents(R_2))$

  - carrega em R_1 o valor na posição de memória obtida ao adicionarmos 100 ao conteúdo do registrador R_2
- Modo de endereçamento útil para ponteiros

Indexando indiretamente

- Podemos também referenciar endereços de forma indireta usando **`*r`** e **`*100(r)`**
- **`LD R1, *100(R2)`**
  `R1 = contents(contents(100 + contents(R2)))`
    - carrega em **`R1`** o valor na posição de memória armazenada na posição de memória obtida ao adicionarmos 100 ao conteúdo de **`R2`**

Carregando constantes

- Podemos também usar constantes como modo imediato de endereçamento, usando **#c**
- **LD R_1, #100**
  - carrega em **R_1** o valor 100
- **ADD R_1, R_1, #100**
  - adiciona 100 ao registrador **R_1**

Exemplo

- `x = y - z`
  - como ficaria a tradução?

Exemplo

```
x = y - z

LD     R1, y
LD     R2, z
SUB    R1, R1, R2
ST     x, R1
```

Exemplo

```
x = y - z

LD      R1, y
LD      R2, z
SUB     R1, R1, R2
ST      x, R1
```

- Podemos melhorar…
- Se y ou z já tivessem sido computadas em um registrador, removeríamos as instruções LD
- Dependendo de como x é usado, podemos evitar armazenar o valor

Acessando *arrays*

```
b = a[i]
// assuma 8 bytes por elemento em a
```

Acessando *arrays*

```
b = a[i]

LD    R1, i          //R1 = i
MUL   R1, R1, 8      //R1 = R1*8
LD    R2, a(R1)      //R2 = c(a+c(R1))
ST    b, R2          //b = R2
```

Acessando *arrays*

```
a[j] = c
```

Acessando *arrays*

```
a[j] = c

LD    R1, c          //R1 = c
LD    R2, j          //R2 = j
MUL   R2, R2, 8      //R2 = R2*8
ST    a(R2), R1      //c(a+c(R2)) = R1
```

Ponteiros

```
x = *p
```

Ponteiros

```
x = *p

LD    R1, p        //R1 = p
LD    R2, 0(R1)    //R2 = c(0+c(R1))
ST    x, R2        //x = R2
```

Ponteiros

```
*p = y
```

Ponteiros

```
*p = y

LD     R1, p          //R1 = p
LD     R2, y          //R2 = y
ST     0(R1), R2   //c(0+c(R1)) = R2
```

Desvios

```
if x < y goto L
```

Desvios

```
if x < y goto L

LD    R1, x          //R1 = x
LD    R2, y          //R2 = y
SUB   R1, R1, R2     //R1 = R1-R2
BLTZ  R1, L          //if R1<0 goto L
```

Custos das instruções

- Geralmente associamos custos ao processo de compilação e execução de um programa
- Dependendo do aspecto que estejamos interessados em otimizar, as medidas variam
  - tempo de compilação, tamanho, tempo de execução, e consumo de energia do programa gerado

Custos das instruções

- Determinar o custo de compilação e execução de um programa é um problema complexo
- Encontrar um programa alvo ótimo para um dado programa fonte é um problema indecidível no geral
- Por simplicidade define-se como o custo de uma instrução: 1 + custo do endereçamento
  - endereçamento com registradores: 0
  - endereçamento com posições de memória: 1

Organização da Memória

- Dividida em quatro áreas
  - *Code*
  - *Static*
  - *Heap*
  - *Stack*

Chamadas de Procedimento

- Para ilustrar geração de código para chamadas de procedimento simplificadas, considere
  - **`call`** *callee*
  - **`return`**
  - **`halt`**
  - **`action`**

Registros de Ativação

- O tamanho e layout dos registros de ativação é determinado pelo gerador de código, de acordo com os nomes na tabela de símbolos

- Por conveniência, assumiremos que a primeira posição guarda o endereço de retorno

- Ilustraremos então como gerar código para chamada do procedimento e retorno do controle após execução

call *callee*

- Implementado por uma sequência de duas instruções de máquina

  ST *callee.staticArea*, **#***here* **+ 20**
  BR *callee.codeArea*

- A primeira instrução salva o endereço de retorno no início do registro de ativação
- A segunda transfere o controle para o código do procedimento sendo chamado

`halt`

- Todo procedimento encerra com uma chamada a return
- Exceto o primeiro procedimento, que não tem *caller*, portanto sua instrução final é halt
    - Devolve o controle ao sistema operacional

`return`

- Implementado por uma instrução de desvio

  BR *****$callee.staticArea$

- Transfere o controle para o endereço salvo no início do registro de ativação de $callee$

```
                    //código de main
action1             //abstrai instrução
call p
action2
halt
                          //código de p
action3
return
```

```
                              //código de main
100:   ACTION1                //código de action1
120:   ST 364, #140 //salva 140 como return
132:   BR 200                 //chama p
140:   ACTION2
160:   HALT                   //retorna ao SO
…
200:   ACTION3
220:   BR *364                //retorna ao endereço 140
…                             //300-363, act. rec. main
300:   …                      //end. de retorno
304:   …                      //dados locais de main
…                             //364-451, act. rec. p
364:   …                      //end. de retorno
368:   …                      //dados locais de p
```

Stack vs. Static Allocation

- Alocação estática pode se tornar dinâmica ao usarmos endereços relativos para armazenamento em registros de ativação

- Em *stack allocation*, a posição de um registro de ativação não é conhecida até execução

- A posição é geralmente armazenada em um registrador, então palavras no registro podem ser acessadas por meio de *offsets*

Registrador **`SP`**

- Registrador que vai conter um ponteiro para o início do registro de ativação no topo da pilha

- Ao chamarmos um procedimento, precisamos alterar o valor de **`SP`** (geralmente consiste de incrementá-lo) e transferir controle

- Após o controle retornar ao *caller*, precisamos novamente alterar o valor de **`SP`** (decrementá-lo), desalocando registro de ativação chamado

Código do main

- Código para o primeiro procedimento inicializa a pilha apontando **`SP`** para o início da área da pilha na memória

```
LD SP, #stackStart
//código do main
HALT
```

Código de chamadas a procedimentos

- Uma chamada de procedimento incrementa **`SP`**, salvando endereço de retorno e transferindo controle

```
ADD SP, SP, #caller.recordSize
ST 0(SP), #here + 16
BR callee.codeArea
```

- O operando **#***caller.recordSize* representa o tamanho do registro de ativação
- O operando **#***here* **`+ 16`** é o endereço da instrução seguinte à **`BR`** *callee.codeArea*

Retorno de procedimentos

- A sequência de retorno consiste de duas partes
- A função chamada transfere controle ao endereço salvo no início do registro de ativação

```
BR *0(SP)
```

- O *caller* decrementa o registrador **SP**, restaurando o valor anterior

```
SUB SP, SP, #caller.recordSize
```

```
                    //código de m
action1
call q
action2
halt
                    //código de p
action3
return
                    //código de q
action4
call p
action5
call q
action6
call q
return
```

```
100:    LD  SP, #600              //inicializa pilha
108:    ACTION1                   //código de action1
128:    ADD SP, SP, #msize        //inicializa pilha
136:    ST  0(SP), #152           //end. de retorno
144:    BR  300                   //call q
152:    SUB SP, SP, #msize        //restaura SP
160:    ACTION2                   //código de action2
180:    HALT                      //retorna ao SO
…                                 //código de p
200:    ACTION3                   //código de action3
220:    BR *0(SP)                 //return
…                                 //código de q
300:    ACTION4                   //tem um desvio condicional para 456
320:    ADD SP, SP, #qsize
328:    ST  0(SP), #344           //end. de retorno na pilha
336:    BR  200                   //call p
344:    SUB SP, SP, #qsize
352:    ACTION5
372:    ADD SP, SP, #qsize
380:    ST  0(SP), #396           //end. de retorno na pilha
388:    BR  300                   //call q
396:    SUB SP, SP, #qsize
404:    ACTION6
424:    ADD SP, SP, #qsize
432:    ST  0(SP), #448           //end. de retorno na pilha
440:    BR  300                   //call q
448:    SUB SP, SP, #qsize
456:    BR *0(SP)                 //return
…
600:    …                         //início da pilha
```