MAB 471
2011.2

Introdução à Geração de Código

http://www.dcc.ufrj.br/~fabiom/comp

Forma do Código

Definição

- Todas as propriedades do código que influenciam no desempenho
- Código em si, abordagens para diferentes construções, armazenamento e codificação de tipos, escolha das operações
- Produto de muitas decisões, várias forçadas pela linguagem

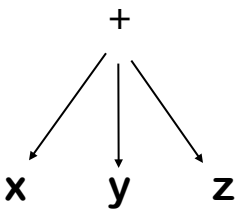
Impacto

- Forma do código influencia algoritmos do compilador
- Forma do código codifica fatos importantes, ou os esconde
- Se estamos falando de geração de código final então a forma do código tem impacto direto no desempenho
- Mas mesmo com um otimizador há limites para o que ele pode fazer

Forma do Código

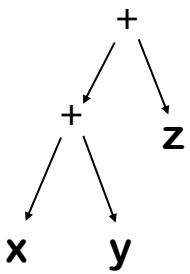
Exemplo

x + y + z



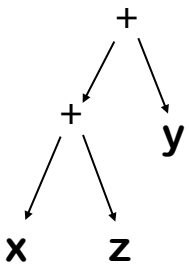
x + y → t1

t1+ z → t2



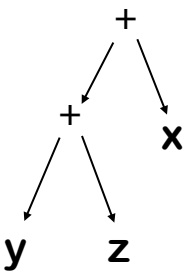
x + z → t1

t1+ y → t2



y + z → t1

t1+ x→ t2



- E se x é 2 e z é 3?
- E se y+z já foi avaliado antes?

A "melhor" forma para x+y+z depende do contexto

Formato do Código

Outro exemplo -- o comando switch

- Implemente como uma sequência de comandos if-then-else
  - Custo depende de onde o caso realmente ocorre
  - O(número de casos)
- Implemente como uma busca binária
  - Precisa de um conjunto denso de condições para buscar
  - Custo (log n) uniforme
- Implemente como uma tabela de saltos
  - Procure endereço na tabela e salte para ele
  - Custo constante uniforma

Compilador tem que escolher melhor estratégia

Nenhum otimizador vai transformar uma forma em outra

Forma do Código

Não dá para confiar no otimizador do back-end?

- Otimizador dá respostas aproximadas para muitos problemas difíceis
- Os passos do compilador têm que ser rápidos
- Frequentemente é benéfico ter um IR com mais informação explícita ("baixo nível")
  - Forma de uma expressão ou estrutura de controle
  - Se um valor está em um registrador ou na memória
- (Re)derivar estas informações pode ser caro
- Codificá-las na IR simplifica e acelera os algoritmos

Gerando Código para Expressões

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(node.left));
          t2← expr(node.right));
          result ← GetTemp();
          emit (op(node), t1, t2, result);
          break;
      case ID:
          t1← base(node);
          t2← offset(node);
          result ← GetTemp();
          emit (loadAO, t1, t2, result);
          break;
      case NUM:
          result ← GetTemp();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
}
```

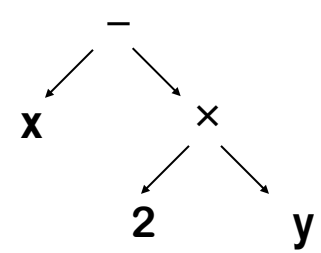
O Conceito

- Assume uma AST como entrada e uma IR linear como saída
- Percorre a árvore em pós-ordem
  - Visita e avalia filhos
  - Emite código para a operação em si
  - Returna temporário com o resultado
- Esconde complexidade do endereçamento de nomes nas rotinas
  - base(), offset(), e val()
- Funciona para expressões simples

Gerando Código para Expressões

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(node.left));
          t2← expr(node.right));
          result ← GetTemp();
          emit (op(node), t1, t2, result);
          break;
      case ID:
          t1← base(node);
          t2← offset(node);
          result ← GetTemp();
          emit (loadAO, t1, t2, result);
          break;
      case NUM:
          result ← GetTemp();
          emit (loadI, val(node), none, result);
          break;
      }
      return result;
 }
```

Exemplo:



Produz:

loadI	@x	⇒ r1
loadAO	r_{arp},r1	⇒ r2
loadI	2	⇒ r3
loadI	@y	⇒ r4
loadAO	r_{arp},r4	⇒ r5
mult	r3,r5	⇒ r6
sub	R2,r6	⇒ r7

Extensões

Casos mais complexos para ID

- E quanto a valores que já estão em registradores?
  - Já em um registrador ⇒ returna o nome do registrador
  - Não está em um reg. ⇒ carrega como antes, mas grava isso
  - Como reutilizar registradores?
- E quanto a argumentos da função?
  - Alguns protocolos de chamada passam args em registradores (x64)
  - Se o protocolo de chamada usa a pilha então argumentos call-by-value são locais como as outras
  - Argumentos call-by-reference (VAR de Pascal, & de C++) têm uma indireção a mais
- E quanto a chamadas de função?
  - Gerar a sequência de chamada e carregar o valor de retorno (em x86 ele vem num registrador)
  - Pode ter impacto grande na geração de código pro resto da expressão

Extensões

Outros operadores

- Avalia os operandos, e faz a operação
- Operações complexas podem virar chamadas pra funções do ambiente de execução
- Atribuição pode ser outro operador

Expressões que misturam tipos (em linguagens estaticamente tipadas)

- Insere conversões de tipos quando necessário
- Análise semântica já garantiu que a operação é válida

Extensões

E quanto à ordem de avaliação?

- Pode usar comutatividade e associativade para melhorar o código
- Bastante difícil (diversas heurísticas)

Mais simples é mudar a ordem de avaliação em uma única op.

- 1º operando tem que ser preservado enquanto 2º é avaliado
- Toma um registrador extra para 2º operando
- Solução: avaliar operando mais complexo primeiro
- Pode-se usar altura da sub-árvore da AST

Atribuição

lhs* ← *rhs

Estratégia

- Avaliar rhs para um valor (um rvalue)
- Avaliar lhs para um local (um lvalue)
  - lvalue é um registrador ⇒ move rhs
  - lvalue é um endereço ⇒ store rhs
- Se rvalue e lvalue têm tipos diferentes
  - Avaliar rvalue para seu tipo "natural"
  - Converte esse valor para o tipo de *lvalue

Booleanos e Expressões Relacionais

Como o compilador deve representá-las?

- Reposta depende da máquina destino

Implementação de booleanos, expressões relacionais e controle de fluxo varia muito entre arquiteturas

Duas abordagens clássicas

- Representação numérica (explícita)
- Representação posicional (implícita)

Qual a melhor depende do contexto e da arquitetura

Booleanos e Expressões Relacionais

Representação numérica

- Dar valores para TRUE e FALSE
- Usar operações AND, OR, e NOT da máquina
- Usar comparações para obter booleano de uma expressão relacional

Exemplos

x < y	vira	cmp_LT	r_x, r_y	$\Rightarrow r_1$
if (x < y) then $stmt_1$ else $stmt_2$	vira	cmp_LT cbr	r_x, r_y r_1	$\Rightarrow r_1$ $\rightarrow$ _$stmt_1$,_$stmt_2$

Booleanos e Expressões Relacionais

E se a arquitetura usa um flag (x86, x64, LVM)?

- Tem que usar salto condicional para interpretar resultado de comparação
- Precisa de saltos na avaliação

Example

x < y	vira		cmp	r_x, r_y	⇒	cc_1
			cbr_LT	cc_1	→	L_T, L_F
		L_T:	loadI	1	⇒	r_2
			br		→	L_E
		L_F:	loadI	0	⇒	r_2
		L_E:	... outros comandos ...			

Essa "representação posicional" é mais complexa

Booleanos e Expressões Relacionais

Representação posicional codifica booleanos no PC

Se o resultado é usado para controlar uma operação tudo bem

Exemplo

if (x < y)
  then a ← c + d
  else a ← e + f

Flags				Booleanos			
	comp	r_x,r_y	⇒ CC_1		cmp_LT	r_x,r_y	⇒ r_1
	cbr_LT	CC_1	→ L_1,L_2		cbr		→ L_1,L_2
L_1:	add	r_c,r_d	⇒ r_a	L_1:	add	r_c,r_d	⇒ r_a
	br		→ L_{OUT}		br		→ L_{OUT}
L_2:	add	r_e,r_f	⇒ r_a	L_2:	add	r_e,r_f	⇒ r_a
	br		→ L_{OUT}		br		→ L_{OUT}
L_{OUT}:	nop			L_{OUT}:	nop		

Versão com flag não produz x < y diretamente

Versão booleana produz

Mas não há muita diferença no código gerado

Booleanos e Expressões Relacionais

Considere a atribuição $x \leftarrow a < b \wedge c < d$

Flags				Booleanos		
	comp	r_a, r_b	$\Rightarrow CC_1$	cmp_LT	r_a, r_b	$\Rightarrow r_1$
	cbr_LT	CC_1	$\rightarrow L_1, L_2$	cmp_LT	r_c, r_d	$\Rightarrow r_2$
L_1:	comp	r_c, r_d	$\Rightarrow CC_2$	and	r_1, r_2	$\Rightarrow r_x$
	cbr_LT	CC_2	$\rightarrow L_3, L_2$			
L_2:	loadI	0	$\Rightarrow r_x$			
	br		$\rightarrow L_{OUT}$			
L_3:	loadI	1	$\Rightarrow r_x$			
	br		$\rightarrow L_{OUT}$			
L_{OUT}:	nop					

Aqui, booleanos produzem código bem melhor

Avaliação de Curto Circuito

Otimizar avaliação de expressões booleanas

- Assim que o valor final é determinado, pule o resto da avaliação

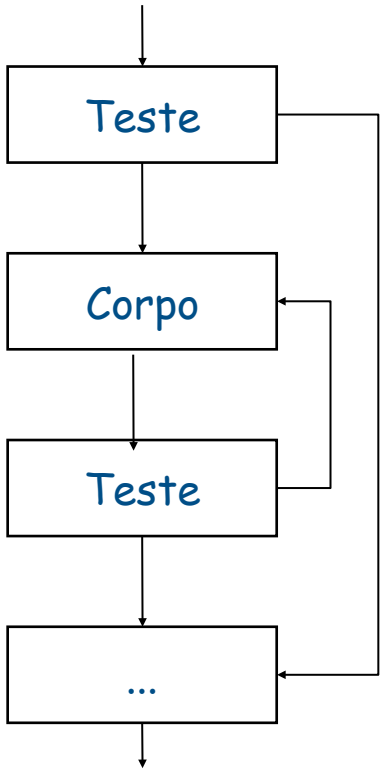
```
if (x or y and z) then …
```

  - Se x é verdadeiro, não precisa avaliar y ou z
    - → Pula direto pra cláusula "then"
    - → Historicamente feito por razões de eficiência
- Não fazer o curto circuito pode ser mais eficiente em arquiteturas modernas
  - Saltos podem custar caro, devem ser evitados se não necessários
  - Mas linguagens ainda podem exigir curto-circuito na sua especificação
  - Compilador pode analisar a expressão para determinar se suprimir curto-circuito é seguro

Controle de Fluxo

Laços

- Avaliar condição antes do laço (se necessário)
- Avaliar condição depois do laço
- Pular de volta pro topo (se necessário)
- Código da condição duplicado



while, for, do, e until se encaixam nesse modelo básico

Break/Continue

Muitas linguagens de programação incluem break/continue

- Sai do laço mais interno (ou do switch)

Break vira um salto para fora do laço

Continue vira um salto para o teste

Só fazem sentido se corpo do loop tem mais de um bloco (por quê?)