Segunda Prova de Análise e otimização de Código
- DCC888 -
Ciência da Computação

Nome: _____
"Eu dou minha palavra de honra que não trapacearei neste exame."

Número de matrícula: _____

As regras do jogo:

- A prova é sem consulta.
- Quando terminar, não entregue nada além do caderno de provas para o instrutor.
- Cada estudante tem direito a fazer uma pergunta ao instrutor durante a prova. Traga o caderno de provas quando vier à mesa do instrutor.
- A prova termina uma hora e quarenta minutos após seu início.
- Seja honesto e lembre-se: **você deu sua palavra de honra**.

Alguns conselhos:

- Escreva sempre algo nas questões, a fim de ganhar algum crédito parcial.
- Você pode sacrificar sua pergunta para saber por que Hilda deixou a vida de princesa para entrar na vida de rameira.
- Se não entender alguma questão, e já tiver gasto sua pergunta, escreva a sua interpretação da questão junto à resposta.
- A prova não é difícil, ela é divertida, então aproveite!

Tabela 1: Pontos acumulados (para uso do instrutor)

Questão 1	Questão 2	Questão 3	Questão 4

Ponto Extra: se você fosse um plutoniano empreendedor, qual seria seu grande empreendimento?

1. (10 Pontos) Considere uma linguagem de expressões aritméticas e booleanas, que possui a seguinte sintaxe e semântica:

```
E ::= E or E
    | E < E
    | zero
    | succ E
    | true
    | false
```

$$\frac{E1 \to E1'}{E1 \text{ or } E2 \to E1' \text{ or } E2}\ [\text{EorE1}] \qquad \frac{E1 \to E1'}{E1 > E2 \to E1' > E2}\ [\text{EgtE1}]$$

$$\frac{E2 \to E2'}{E1 \text{ or } E2 \to E1 \text{ or } E2'}\ [\text{EorE2}] \qquad \frac{E2 \to E2'}{E1 > E2 \to E1 > E2'}\ [\text{EgtE2}]$$

$$\text{true or false} \to \text{true}\ [\text{EorTF}] \qquad \text{succ N} > \text{zero} \to \text{true}\ [\text{EgtSZ}]$$

$$\text{false or true} \to \text{true}\ [\text{EorFT}] \qquad \text{zero} > \text{succ N} \to \text{false}\ [\text{EgtZS}]$$

$$\text{false or false} \to \text{false}\ [\text{EorFF}] \qquad \text{true or true} \to \text{true}\ [\text{EorTT}]$$

Podemos definir o *tamanho* de um termo da seguinte maneira:

$$\text{size(zero)} = 1\ [\text{SZr}] \qquad \text{size(true)} = 1\ [\text{STr}] \qquad \text{size(false)} = 1\ [\text{SFl}]$$

$$\text{size(E1 or E2)} = \text{size(E1)} + \text{size(E2)} + 1\ [\text{SOr}]$$

$$\text{size(E1 > E2)} = \text{size(E1)} + \text{size(E2)} + 1\ [\text{SGt}]$$

Prove o seguinte teorema: se E é um termo sintaticamente válido e existe uma regra de avaliação r tal que $E \to E'$, então $size(E) > size(E')$.

2. (10 Pontos) Considere a linguagem de expressões aritméticas e booleanas vista na questão anterior. As seguintes regras de tipagem são definidas para essa nossa linguagem:

$$\text{T ::= Bool} \mid \text{Nat}$$

$$\text{zero} : \text{Nat} \quad [\text{TZr}] \qquad \frac{\text{E1} : \text{Nat} \quad \text{E2} : \text{Nat}}{\text{E1} > \text{E2} : \text{Bool}} \quad [\text{TGt}]$$

$$\text{true} : \text{Bool} \quad [\text{TTr}] \qquad \frac{\text{E1} : \text{Bool} \quad \text{E2} : \text{Bool}}{\text{E1 or E2} : \text{Bool}} \quad [\text{TOr}]$$

$$\text{false} : \text{Bool} \quad [\text{TFl}] \qquad \frac{\text{E} : \text{Nat}}{\text{succ E} : \text{Nat}} \quad [\text{TSc}]$$

Prove que a linguagem possui a propriedade de *progresso*. Em outras palavras, se E é um termo sintaticamente válido, tal que E possui tipo T de acordo com alguma das regras acima, então existe uma regra de avaliação (dentre aquelas vistas na questão anterior), tal que $E \rightarrow E'$ **ou** o termo E é um valor.

3. A alocação de registradores em programas no formato SSA tem algumas vantagens sobre a alocação no formato tradicional. Uma dessas vantagens é que um programa em formato SSA nunca necessita de mais registradores que o programa que lhe deu origem. Em outras palavras, se P é um programa, e P_{ssa} é sua versão em formato SSA, então $\text{MinReg}(P) \geq \text{MinReg}(P_{ssa})$, sendo MinReg o menor número de registradores necessário para compilar o programa. O restante dessa questão refere-se a essa vantagem da alocação de registradores em programas no formato SSA.

   (a) (4 Pontos) $\text{MinReg}(P) \geq \text{MinReg}(P_{ssa})$ é uma desigualdade estrita. Em outras palavras, existem programas P tais que $\text{MinReg}(P) = \text{MinReg}(P_{ssa})$. Demonstre esse fato mostrando um programa para o qual a igualdade é verdadeira. Escreva o programa original, e o mesmo programa em formato SSA. Apresente um argumento explicando porque eles precisam do mesmo número de registradores.

   (b) Existem também programas P tais que $\text{MinReg}(P) > \text{MinReg}(P_{ssa})$. Demonstre tal fato escrevendo um programa que possua essa propriedade. Mostre:

      i. (2 Pontos) O programa original.
      ii. (2 Pontos) O programa em formato SSA.
      iii. (1 Ponto) O programa original após a alocação de registradores.
      iv. (2 Ponto) O programa em formato SSA após a alocação de registradores, e após a eliminação de funções ϕ. Você pode usar cópias simples, ou trocas (`swap_regs(R1, R2)`), para implementar as funções ϕ.

4. Considere uma análise de código que encontre, para cada variável v, o conjunto das variáveis menores que v. Definimos o estado abstrato de uma variável v como $[\![v]\!] = \{v_1, v_2, \ldots, v_n\}$, onde cada v_i é definitivamente menor ou igual a v.

   (a) Defina um semi-reticulado $(S, <, \wedge, \top, \bot)$ para representar essa análise. Você deve indicar:

      i. (1 Ponto) Qual conjunto S você está usando
      ii. (1 Ponto) O que significa o operador de ordem parcial $<$
      iii. (1 Ponto) O operador do operador de junção (*meet*) $\wedge$
      iv. (1 Ponto) O elemento $\top$ que está presente em S. Lembre-se $\top \wedge x = x$
      v. (1 Ponto) O elemento $\bot$ que está presente em S. Lembre-se $\bot \wedge x = \bot$

   (b) (5 Pontos) Considere, agora, uma linguagem de programação muito simples, com os quatro tipos de instrução abaixo:

   - `assign`$(v,\ u)$, copia o valor de u em v. Note que u pode ser uma constante.
   - `add`$(v,\ v_1,\ v_2)$. Atribui a v o valor de $v_1 + v_2$.
   - `if`(v_1, v_2, l). Se v_1 for menor que v_2, então desvia o fluxo de execução para l.
   - `phi`$(v, v_1, v_2, \ldots, v_n)$. Função $v = \phi(v_1, v_2, \ldots, v_n)$.

   Queremos construir uma análise esparsa. Assim, considere uma representação intermediária que divida a linha de vida de variáveis usadas em condicionais. Em vez de `if`(v_1, v_2, l), passamos a ter `if_sparse`$(v_1, v_2, l : v_1', v_2', v_1", v_2")$. Essa instrução copia v_1, v_2 para v_1', v_2' se o desvio for tomado, e copia v_1, v_2 para $v_1", v_2"$ caso contrário. Defina funções de transferência para essa análise esparsa. Você precisa definir uma função para cada uma das quatro instruções de nossa linguagem. Cada uma dessas funções preenche o estado abstrato de uma ou mais variáveis de nossa linguagem. Lembre-se de considerar `if_sparse` em vez de `if`.