

TEORIA DA COMPUTAÇÃO

2ª Prova - Resumo

Nikolas Machado Corrêa

Redutibilidade: principal método para provar que problemas são computacionalmente insolúveis.

Redução: maneira de converter um problema em outro de forma que a solução para o segundo possa ser usada para resolver o primeiro. Exemplo: orientar-se em uma cidade (A) pode ser **reduzido** ao problema de se obter um mapa (B).

→ Se A se **reduz** a B, podemos encontrar solução para B para resolver A.

- Se A é **reduzível** a B, resolver A não pode ser mais difícil que resolver B, uma vez que ao encontrar solução para B, resolve-se A.

Se A for reduzível a B e B for decidível, A também será decidível.
Equivalentemente, se A for indecidível e reduzível a B, B será indecidível.

Provar indecidibilidade de um problema: mostrar que outro indecidível se reduz a ele.

Problemas Indecidíveis

Problema de Aceitação:

A(MT) = problema de determinar se uma MT aceita uma dada entrada → indecidível

Problema da Parada:

PARA(MT) = problema de determinar de uma MT para, dada uma entrada.

Ideia: usar indecidibilidade de $A(MT)$ para provar que $PARA(MT)$ é insolúvel, reduzindo $A(MT)$ a $PARA(MT)$.

Teorema: $PARA(MT)$ é indecidível.

Ideia da prova: provar por contradição, isto é, supõe-se que $PARA(MT)$ é decidível e usa-se essa suposição para mostrar que $A(MT)$ também é decidível (contradição!).

Prova Formal: supondo que uma MT R decida $PARA(MT)$. Constrói-se uma MT S para decidir $A(MT)$, operando da seguinte forma:

1. Rode a MT R , dando como entrada uma MT M e uma cadeia w ;
2. Se R rejeita, rejeite. Se R aceita, simule M sobre w até que ela pare;
3. Se M aceitou, aceite; caso contrário, rejeite.

Se R decide $PARA(MT) \rightarrow S$ decide $A(MT)$

Como $A(MT)$ é indecidível $\rightarrow PARA(MT)$ também é.

$$V_{MT} = \{\langle M \rangle \mid M \text{ eh uma MT e } L(M) = \emptyset\}$$

Teorema: $V(MT)$ é indecidível.

$M_1 =$ "Sobre a entrada x :

1. Se $x \neq w$, rejeite.
2. Se $x = w$, rode M sobre a entrada w e aceite se M aceita."

Prova: Supomos que uma MT R decide $V(MT)$, constrói-se uma MT S que decide $A(MT)$. Assim, Usa-se a descrição de M e w para construir uma MT M_1 ;

1. Roda-se R sobre M_1 ;
2. Se R aceita, rejeite; se R rejeita, aceite;

Cálculo λ

Cálculo Lambda: baseado na ideia de que algoritmos podem ser vistos como funções matemáticas mapeando inputs para outputs. Todas as computações são reduzidas a operações básicas de definição de funções e aplicação.

Máquina de Turing \rightarrow modelo para linguagens imperativas

Cálculo Lambda \rightarrow modelo para linguagens funcionais

1. a função $f : x \mapsto x + y$ é escrita como $\lambda x. x + y$.
2. a função $g : y \mapsto x + y$, é escrita como $\lambda y. x + y$.
3. a função $h : x, y \mapsto x + y$, é escrita como $\lambda xy. x + y$.

A sintaxe do Cálculo Lambda tem 3 tipos de termos:

1. **Variável** $x \rightarrow$ termo
2. **Abstração** de uma variável x de um termo $t_1 \rightarrow \lambda x. t_1$
3. **Aplicação** de um termo t_1 a um termo $t_2 \rightarrow$ é também um termo

BNF (Backus Normal Form):

$t ::=$	termos
x	variável
$\lambda x. t$	abstração
$t \ t$	aplicação

<i>Sintaxe concreta</i>	<i>Sintaxe abstrata</i>
strings de caracteres escritas diretamente	representação interna de programas como árvores com labels (abstract syntax trees ou AST)

Transformação concreta \rightarrow abstrata:

1. **Lexer:** converte a string de caracteres em sequência de tokens - identificadores, palavras-chave, constantes, etc. Remove comentários e trata de assuntos como espaços em branco.
2. **Parser:** transforma essa sequência de tokens em árvores de sintaxe abstrata. Convenções como precedência e associatividade reduzem necessidades de parênteses.

Convenções:

1. Aplicação associa à esquerda. Exemplo: $s\ t\ u$ é o mesmo que $(s\ t)\ u$
2. Corpo de uma abstração estende máximo possível à direita

Conjunto de variáveis livres:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x. t_1) &= FV(t_1) - \{x\} \\ FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

Cada passo na computação consiste de reescrever uma aplicação cujo componente esquerdo é uma abstração, substituindo a variável ligada no corpo da abstração pelo componente do lado direito:

$$(\lambda x. t_{12})\ t_2 \rightarrow [x \mapsto t_2]\ t_{12},$$

$[x \mapsto t_2]\ t_{12}$ = termo obtido pela substituição de todas as ocorrências livres de x em t_{12} por t_2

Termo da forma $(\lambda x. t_{12})\ t_2$ é chamado **redex**. **Redução-beta** (beta-reduction) é reescrever um redex de acordo com a regra acima.

Estratégias de Avaliação: definem qual redex em um termo pode ser reduzido no próximo passo de avaliação:

-
1. **Full beta-reduction:** Qualquer redex pode ser reduzido a qualquer momento. Falta de uma estratégia de redução específica;
 2. **Normal order:** Redex mais à esquerda e mais externo é sempre reduzido primeiro. Ou seja, sempre que possível, os argumentos são substituídos no corpo de uma abstração antes que sejam reduzidos;
 3. **Call-by-need:** Chamado em contextos práticos "avaliação preguiçosa" (lazy evaluation);
 4. **Call-by-value:** Apenas redexes mais externos são reduzidos (somente quando seu lado direito é reduzido a um valor (variável ou abstração);
 5. **Call-by-name:** Como *Normal Order*, exceto que nenhuma redução é realizada dentro de abstrações;