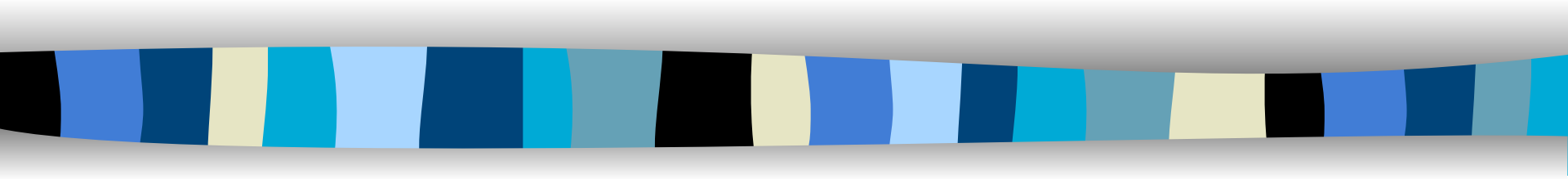
Técnicas para Avaliação Parcial de Programas



Vladimir Oliveira Di Iorio

Avaliador Parcial

- prog. *P* tem duas entradas (in_1 e in_2);
- avaliador parcial recebe *P* e in_1, produzindo um novo programa P_{in_1};
- execução de P_{in_1} sobre in_2 produz mesmos resultados que *P* sobre in_1 e in_2.
- P_{in_1} : programa *residual* ou *especializado.*
- in_1 e in_2 : entradas *estática* e *dinâmica*.
- Objetivo: **eficiência!**

Exemplo

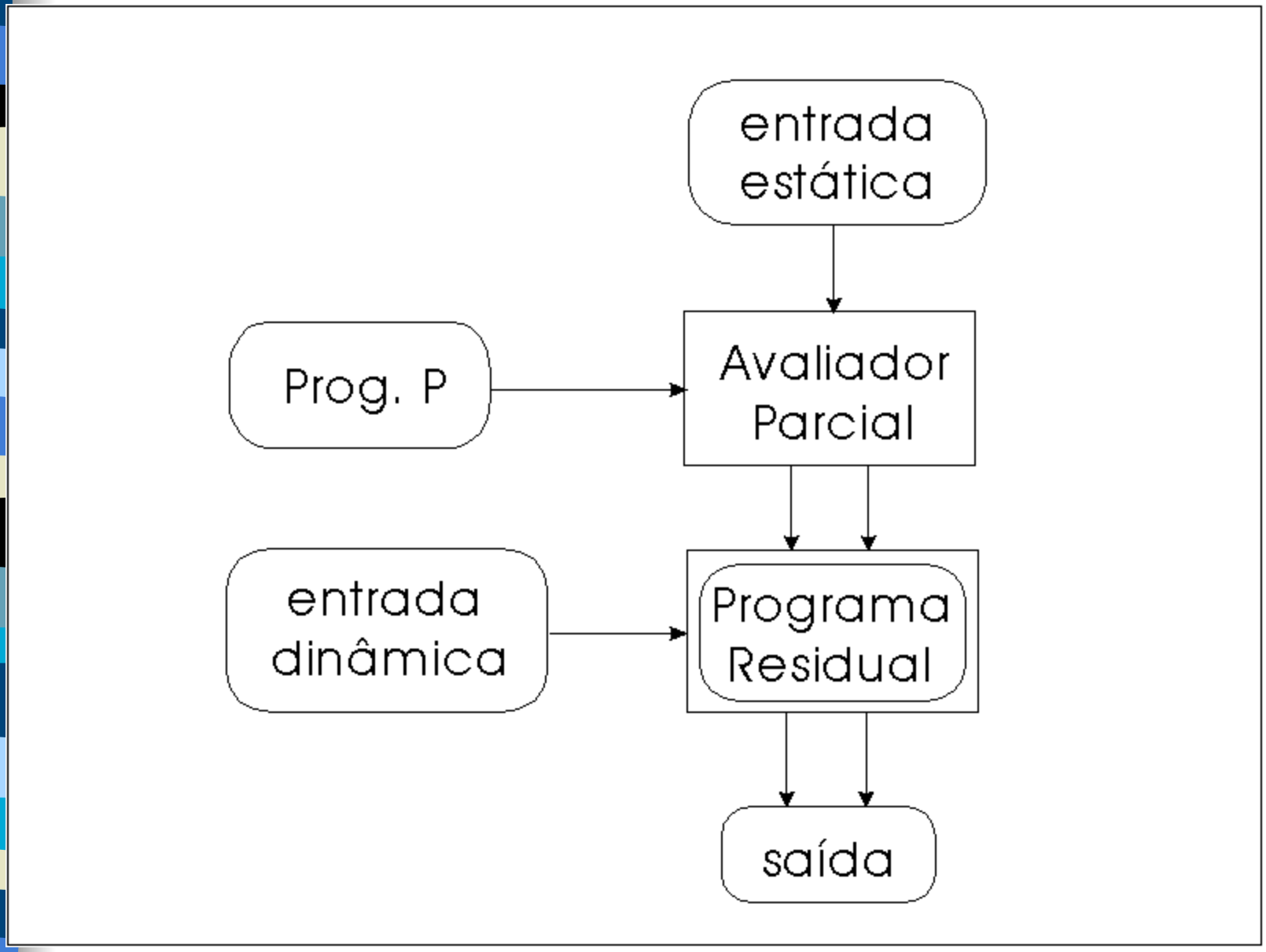
```
Int Power (int n, int x) {
    int p = 1;
    while (n > 0)
        if (n%2 == 0) {
            x = x * x;
            n = n / 2;
        }
        else {
            p = p * x;
            n = n - 1;
        }
    return p;
}
```

```
// especialização
// com n = 5:

int Power_5 (int x) {
    int p  = x;
    x = x * x;
    x = x * x;
    p = p * x;
    return p;
}
```

Aplicações de Avaliação Parcial

- Programas genéricos X eficientes
- Computação gráfica
- Simulação de circuitos
- Casamento de Padrões
- Compilação e Geração de Compiladores
- etc


entrada
estática
Prog. P
Avaliador
Parcial
entrada
dinâmica
Programa
Residual
saída

MIX

- Como é feita a avaliação parcial?
  - Executando os cálculos que dependem apenas da entrada estática.
  - Gerando código para as estruturas que dependem da entrada dinâmica.
- Mistura *execução* + *geração de código*.
- Avaliador parcial é chamado de *MIX*.
- Técnica: *Especialização Polivariante.*
- Abordagens: *Online* e *Offline*.

Linguagem de Fluxograma FCL

- Programa:
  - variáveis de entrada
  - rótulo inicial
  - blocos básicos

Blocos:

rótulo: *atribuição*

...

atribuição

desvio

Desvios:

- *goto* rótulo
- *if* condição *goto* rótulo1 *else* rótulo2
- *return* valor

Exemplo FCL

```
Int Power (int n, int x) {
    int p = 1;
    while (n > 0)
        if (n%2 == 0) {
            x = x * x;
            n = n / 2;
        }
        else {
            p = p * x;
            n = n - 1;
        }
    return p;
}
```

```
(n x)
(b0)
b0: p :=1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

Execução

```
(n x)
(b0)
b0: p := 1
      goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
      n := n / 2
      goto b1
b4: p := p * x
      n := n - 1
      goto b1
b5: return p
```

Para n=2, x=5

Estado = (rótulo,var)
= (rótulo, [n,x,p])

(b0, [2,5,0])
→ (b1, [2,5,1])
→ (b2, [2,5,1])
→ (b3, [2,5,1])
→ (b1, [1,25,1])
→ (b2, [1,25,1])
→ (b4, [1,25,1])
→ (b1, [0,25,25])
→ (b5, [0,25,25])
→ ((halt,25), [0,25,25])

Especialização

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

Especialização (n=5):

```
(x)
(b0)
b0: p := x
    x := x * x
    x := x * x
    p := p * x
    return p
```

Especialização Polivariante

- Programa visto como um grafo:
  - vértices = pontos de programa
  - arestas = fluxo de controle
- Consiste em:
  - determinação dos estados alcançáveis;
  - especialização dos pontos de programa;
  - compressão das transições.
- Abordagens: *Online* e *Offline*.

Abordagem *Online*

- Valores das expressões: (type, val)
  - (S, cte) : valores estáticos
  - (D, exp) : valores dinâmicos
- Inicialização:
  - entrada dinâmica: (D, var)
  - demais variáveis: (S, cte)
- Exemplo: especializar *Power* com $n = 2$.

Determinar Estados Alcançáveis

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

Inicialização:

(b0, [n, x, p])

(b0, [(S,2), (D,x), (S,0)])

(b0, [2,D,0])

→ (b1, [2,D,1])

→ (b2, [2,D,1])

→ (b3, [2,D,1])

Resposta: não faz nada!! Significa que o programa residual tem um loop.

→ (b1, [0,D,D])

→ (b5, [0,D,D])

→ (halt, [0,D,D])

Especializar Pontos de Programa

```
(n x)
(b0)
b0: p := 1
      goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
      n := n / 2
      goto b1
b4: p := p * x
      n := n - 1
      goto b1
b5: return p
```

```
(b0, [2,D,0]):
    goto (b1, [2,D,1])

(b1, [2,D,1]):
    goto (b2, [2,D,1])

(b2, [2,D,1]):
    goto (b3, [2,D,1])

(b3, [2,D,1]):
    x := x * x
    goto (b1, [1,D,1])
```

Especializar Pontos de Programa

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

```
(b1, [1,D,1]):
    goto (b2, [1,D,1])

(b2, [1,D,1]):
    goto (b4, [1,D,1])

(b4, [1,D,1]):
    p := 1 * x
    goto (b1, [0,D,D])

(b1, [0,D,D]):
    goto (b5, [0,D,D])

(b5, [0,D,D]):
    return p
```

Comprimir Transições

(x)
((b0, [2,D,0]))
(b0, [2,D,0]): goto (b1, [2,D,1])
(b1, [2,D,1]): goto (b2, [2,D,1])
(b2, [2,D,1]): goto (b3, [2,D,1])
(b3, [2,D,1]): x := x * x
goto (b1, [1,D,1])
(b1, [1,D,1]): goto (b2, [1,D,1])
(b2, [1,D,1]): goto (b4, [1,D,1])
(b4, [1,D,1]): p := 1 * x
goto (b1, [0,D,D])
(b1, [0,D,D]): goto (b5, [0,D,D])
(b5, [0,D,D]): return p

(x)
(b0)

b0: x := x * x
p := 1 * x
return p

Compressão de Transições

- Opções:
  - *on the fly*
  - *postpass*
- Problemas:
  - duplicação de código
  - *loop* infinito

Outras Otimizações

(x)

(b0)

b0: x := x * x

p := 1 * x

return p

←

…*usando propriedades algébricas*...

(x)

(b0)

b0: x := x * x

p := x

return p

Outras Otimizações

(x)

(b0)

b0: x := x * x

p := x ←

return p ←

…*expandindo*

expressões...

(x)

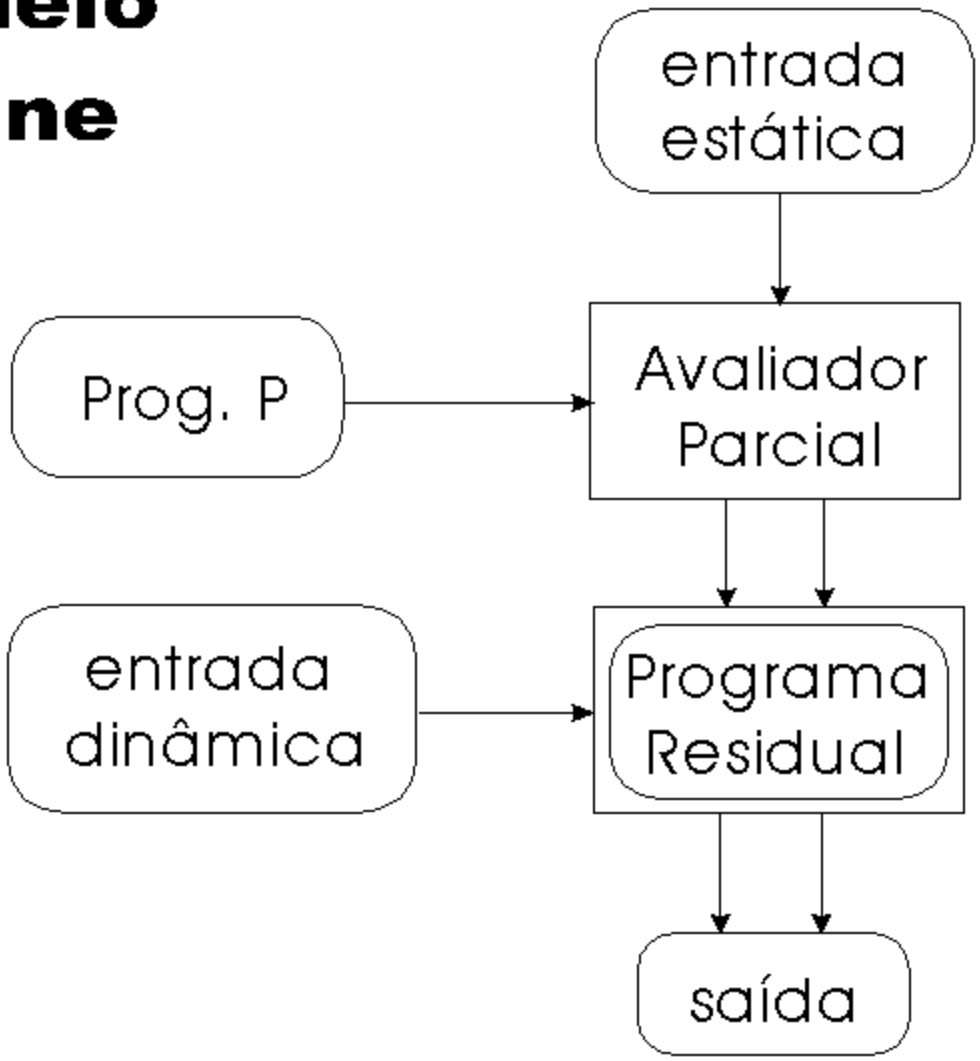
(b0)

b0: return x * x

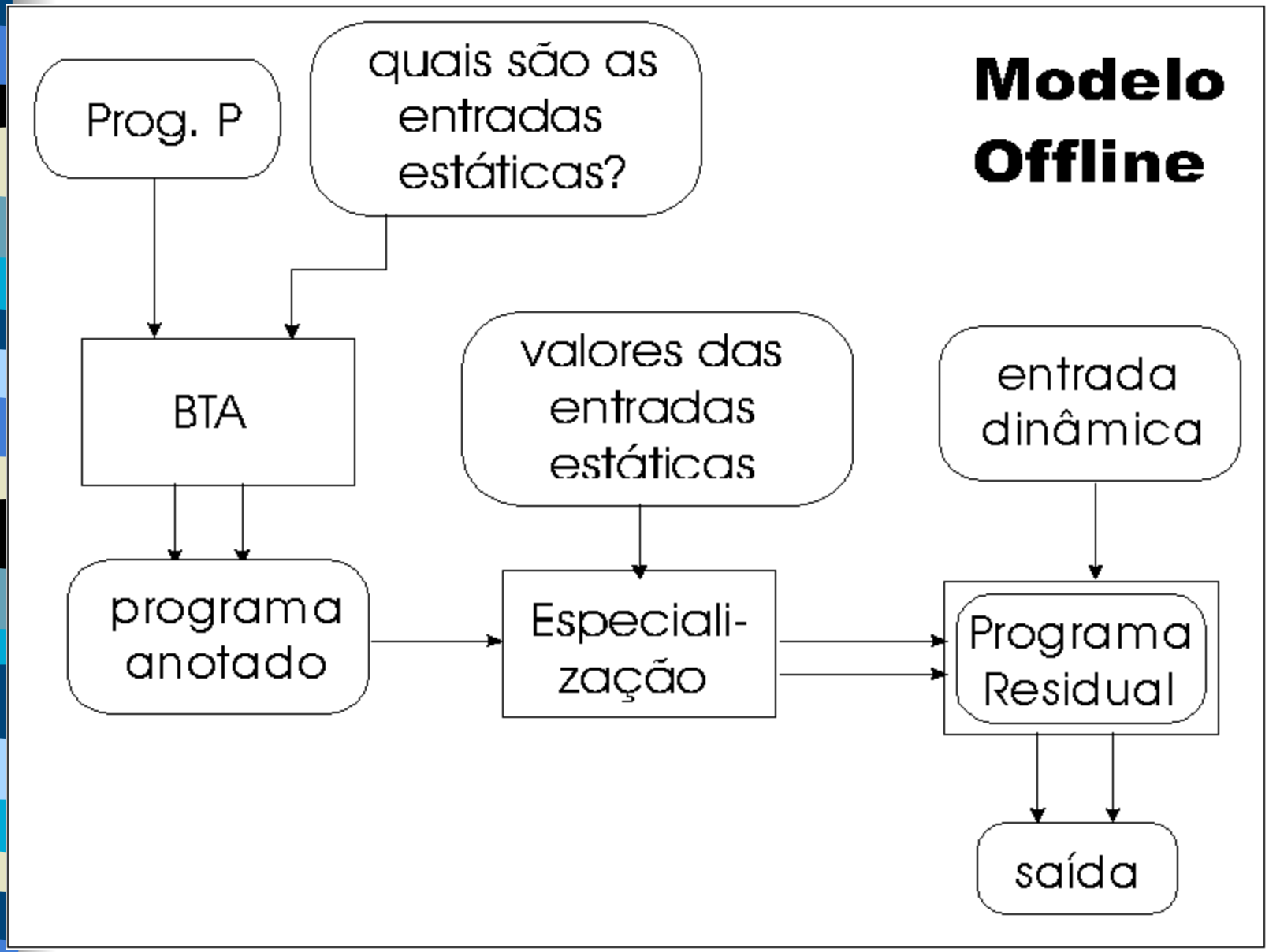
Abordagem *Offline*

- Especialização em 2 passos.
- Análise de Tempo de Definição (BTA):
  - variáveis são classificadas como S ou D;
  - gera programa anotado.
- Especialização:
  - usa valores das variáveis estáticas;
  - segue anotações da BTA para produzir programa residual.

Modelo Online



Modelo Offline



Análise de Tempo de Definição

- Entradas:
  - programa fonte;
  - quais variáveis são estáticas.
- Saídas:
  - classificação de todas as variáveis;
  - programa anotado.
- Algoritmo: iteração até atingir ponto fixo.
- Utiliza *Princípio da Congruência*.

BTA

***Pergunta**:*

término

garantido?

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

← **x é dinâmico, logo p tem que ser dinâmico!**

Supondo:
n estático
x dinâmico

Divisão inicial:
$[n \mapsto S, x \mapsto D, p \mapsto S]$

Primeira iteração:
$[n \mapsto S, x \mapsto D, p \mapsto D]$

Segunda iteração:
$[n \mapsto S, x \mapsto D, p \mapsto D]$

Fim da classificação!

Programa Anotado

Divisão:
$[n \mapsto S, x \mapsto D, p \mapsto D]$

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

Determinar Estados Alcançáveis

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

Inicialização:
(b0, [n])
(b0, [2])

(b0, [2])
→ (b1, [2])
→ (b2, [2])
→ (b3, [2])
→ (b1, [1])
→ (b2, [1])
→ (b4, [1])
→ (b4, [0])
→ (b1, [0])
→ (b5, [0])
→ (halt, [0])

Especializar Pontos de Programa

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

```
(b0, [2]):
    p := 1
    goto (b1, [2])

(b1, [2]):
    goto (b2, [2])

(b2, [2]):
    goto (b3, [2])

(b3, [2]):
    x := x * x
    goto (b1, [1])
```

Especializar Pontos de Programa

```
(n x)
(b0)
b0: p := 1
    goto b1
b1: if n > 0 goto b2 else b5
b2: if n%2 = 0 goto b3 else b4
b3: x := x * x
    n := n / 2
    goto b1
b4: p := p * x
    n := n - 1
    goto b1
b5: return p
```

```
(b1, [1]):
    goto (b2, [1])

(b2, [1]):
    goto (b4, [1])

(b4, [1]):
    p := p * x
    goto (b1, [0])

(b1, [0]):
    goto (b5, [0])

(b5, [0]):
    return p
```

Comprimir Transições

(b0, [2]): p := 1
goto (b1, [2])

(b1, [2]): goto (b2, [2])

(b2, [2]): goto (b3, [2])

(b3, [2]): x := x * x
goto (b1, [1])

(b1, [1]): goto (b2, [1])

(b2, [1]): goto (b4, [1])

(b4, [1]): p := p * x
goto (b1, [0])

(b1, [0]): goto (b5, [0])

(b5, [0]): return p

(x)
(b0)

b0: p := 1
x := x * x
p := p * x
return p

…*online*:
(x)
(b0)

b0: x := x * x
p := 1 * x
return p

Comparação *Online* X *Offline*

- *Offline*: mais conservador.
- *Online*: propagação de *tags*.
- Analogia com verificação de tipos:
  - *online* análogo a verificação dinâmica;
  - *offline* análogo a verificação estática.
- Primeiros avaliadores: *online*.
- Compilação e geração de compiladores com auto-aplicação: *offline*!

Tópicos Mais Avançados

- Divisão Monovariante Congruente: classificação das variáveis vale para o programa inteiro.
- Algumas vezes, pode gerar loop infinito.
- Outras divisões mais complexas:
  - Divisão *Pointwise*.
  - Divisão *Polivariante*.

Exemplo: Máquina de Turing

- Instruções:
  - right
  - left
  - write *a*
  - goto *L*
  - if *a* goto *L*

- Programa exemplo:

0: if 0 goto 3

1: right

2: goto 0

3: write 1

Interpetador para MT

- Programa:

```
   0: if 0 goto 3
→  1: right
   2: goto 0
   3: write 1
```

var *prog*:

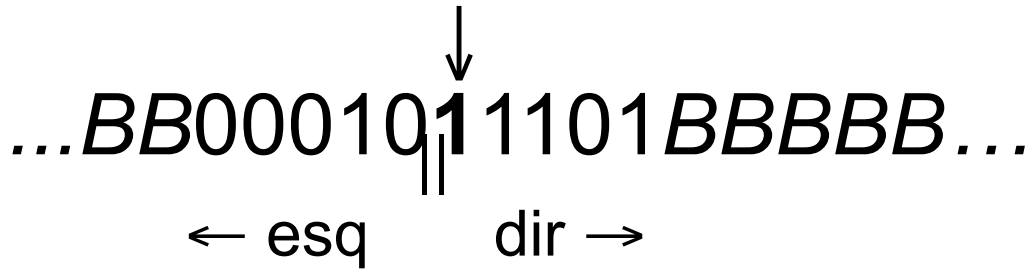
```
[ [0 ‘ifgoto’ 0
     3]
  [1 ‘right’]
  [2 ‘goto’ 0]
  [3 ‘write’ 1]
]
```

var *progrest*:

```
[ [1 ‘right’]
  [2 ‘goto’ 0]
  [3 ‘write’ 1]
]
```

Interpretador para MT

- Fita:



variável *esq*: [0 1 0 0 0]

variável *dir*: [1 1 1 0 1]

```
(prog, dir)     ←  Entradas para o Interpretador:
                   • prog é um programa MT;
(init)             • dir é o estado inicial da fita.
init:   progrest := prog
        esq := [ ]
        goto loop
loop:   if progrest = [ ] goto stop else cont
cont:   instr := hd (progrest)
        progrest := tl (progrest)
        op := hd (tl (instr))
        if op = ‘right’ goto do-right else
cont1:  cont1
cont2:  if op = ‘left’ goto do-left else cont2
cont3:  if op = ‘write’ goto do-write else
        cont3
stop:   if op = ‘goto’ goto do-goto else do-if

        return dir
```

```
do-right:  esq := cons (hd(dir), esq)
           dir := tl (dir)
           goto loop
do-left:   dir := cons (hd(esq), dir)
           esq := tl (esq)
           goto loop
do-write:  simb := hd (tl (tl (instr)))
           dir := cons (simb, tl(dir))
           goto loop
do-goto:   prox := hd (tl (tl (instr)))
           goto jump
do-if:     simb := hd (tl (tl (instr)))
           prox := hd (tl (tl (tl (instr))))
           if simb = hd(dir) goto jump else loop
jump:      progrest := proxintsr (prox, prog)
           goto loop
```

BTA

- Especializar interpretador com respeito a um programa MT específico:
  - *prog* estático;
  - *dir* dinâmico.
- Divisão inicial:

$$[\ \mathrm{prog} \mapsto S, \mathrm{dir} \mapsto D, \mathrm{progrest} \mapsto S, \mathrm{esq} \mapsto S, \mathrm{instr} \mapsto S, \mathrm{op} \mapsto S, \mathrm{simb} \mapsto S, \mathrm{prox} \mapsto S\ ]$$

- Primeira iteração:

```
...
do-right:  esq := cons (hd(dir), esq)
           dir := tl (dir)
           goto loop
```

← *esq depende de dir*

- Nova divisão:

[prog ↦ S, dir ↦ D, progrest ↦ S, esq ↦ D,
instr ↦ S, op ↦ S, simb ↦ S, prox ↦ S]

- Segunda iteração: nenhuma alteração!

```
(prog, dir)
(init)
init:    progrest := prog
         esq := []      ← Lift
         goto loop
loop:    if progrest = [ ] goto stop else cont
cont:    instr := hd (progrest)
         progrest := tl (progrest)
         op := hd (tl (instr))
         if op = ‘right’ goto do-right else
cont1:   cont1
cont2:   if op = ‘left’ goto do-left else cont2
cont3:   if op = ‘write’ goto do-write else
         cont3
stop:    if op = ‘goto’ goto do-goto else do-if

         return dir
```

```
do-right:  esq := cons (hd(dir), esq)
           dir := tl (dir)
           goto loop
do-left:   dir := cons (hd(esq), dir)
           esq := tl (esq)
           goto loop
do-write:  simb := hd (tl (tl (instr)))
           dir := cons (simb, tl(dir))
           goto loop
do-goto:   prox := hd (tl (tl (instr)))
           goto jump
do-if:     simb := hd (tl (tl (instr)))
           prox := hd (tl (tl (tl (instr))))
           if simb = hd(dir) goto jump else loop
jump:      progrest := proxintsr (prox, prog)
           goto loop
```

Otimização

- Análise de variáveis vivas:

```
...
do-write: simb := hd (tl (tl (instr)))      ← variável simb definida
          dir := cons (simb, tl(dir))       ← variável simb utilizada
          goto loop

loop: if progrest = [ ] goto stop else cont ← variável simb morta
...
```

Especialização

var *prog* = [[0 ‘ifgoto’ 0 3]
[1 ‘right’]
[2 ‘goto’ 0]
[3 ‘write’ 1]]

Init: progrest := prog
esq := [ ]
goto loop

- Variáveis vivas: apenas *prog.*
- *prog* não muda: fora da representação.
- Estado inicial: *(init, [ ])*

(Init, [ ]): esq := [ ]
goto (loop, [progrest $\mapsto$ 0]) ← representação simplificada de *progrest*

(loop, [progrest ↦ 0]): …?

loop: if progrest = [ ] goto stop else cont

(loop, [progrest ↦ 0]):
goto (cont , [progrest ↦ 0])

cont: instr := hd (progrest) // … *[0 'ifgoto' 0 3]*
progrest := tl (progrest)
op := hd (tl (instr)) // … *'ifgoto'*
if op = 'right' goto do-right else cont1
cont1: if op = 'left' goto do-left else cont2
cont2: if op = 'write' goto do-write else cont3
cont3: if op = 'goto' goto do-goto else do-if
(cont , [progrest ↦ 0]):
goto (do-if, [progrest ↦ 1, instr ↦ [0 'ifgoto' 0 3]])

(do-if, [progrest ↦ 1, instr ↦ [0 ‘ifgoto’ 0 3]]): …?

```
do-if:  simb := hd (tl (tl (instr)))          // … 0
        prox := hd (tl (tl (tl (instr))))     // … 3
        if simb = hd(dir) goto jump else loop
jump: progrest := proxintsr (prox, prog)
        goto loop
```

(do-if, [progrest ↦ 1, instr ↦ [0 ‘ifgoto’ 0 3]]):

```
    if 0 = hd(dir) goto
        (loop, [progrest ↦ 3]) ←
    else
        (loop, [progrest ↦ 1]) ←
```

Dois novos estados a serem processados!

(loop, [progrest ↦ 3]) processa *[[3 ‘write’ 1]]*

(loop, [progrest ↦ 1]) processa *[[1 ‘right’] [2 ‘goto’ 0] …]*

Loop no Programa Residual

(do-goto, [progrest $\mapsto$ 3, instr $\mapsto$ [2 ‘goto’ 0]]): …?

```
do-goto: prox := hd (tl (tl (instr)))    // … 0
         goto jump
jump:    progrest := proxintsr (prox, prog)
         goto loop
```

(do-goto, [progrest $\mapsto$ 3, instr $\mapsto$ [2 ‘goto’ 0]]):
goto (loop, [progrest $\mapsto$ 0]) $\leftarrow$ *O estado (loop, [progrest $\mapsto$ 0]) já foi gerado!!!*

Programa Residual

```
(dir)
(b0)
b0: esq := [ ]
    goto b1
b1: if 0 = hd(dir) goto b2 else b3
b2: dir := cons (1, tl(dir))
    return dir
b4: esq := cons (hd(dir), esq)
    dir := tl (dir)
    goto b1
```

```
var prog:

[ [0 'ifgoto' 0
     3]
  [1 'right']
  [2 'goto' 0]
  [3 'write' 1]
]
```

- Qual a relação entre o programa MT e o programa MT especializado?
- Interpretador foi especializado com relação a programa MT específico?

Compilação

- Nosso avaliador parcial especializa programas escritos em FCL, com programas residuais em FCL.
- O interpretador de MT foi especializado com relação a um programa MT *P*.
- A semântica do programa residual é a mesma de *P*.
- Especialização = **compilação** da linguagem MT para linguagem FCL.

Definição Equacional de MIX

$$out = [\![P]\!]_S\ (in_1, in_2) \quad \leftarrow$$

$$P_{in_1} = [\![mix]\!]_L\ (P, in_1)$$

$$out = [\![P_{in_1}]\!]_T\ (in_2)$$

[P] é a *semântica* do programa *P*.

S é a linguagem em que *P* está escrito.

Pergunta: nos nossos exemplos, que linguagens são *S*, *L* e *T* ?

Primeira Projeção de Futamura

- Suponha *IntL* um interpretador para uma linguagem *L*:

$$out = [\![IntL]\!](P, input)$$

$$\text{target} = [\![mix]\!](IntL, P)$$

$$out = [\![\text{target}]\!](input)$$

- Compilação por meio da avaliação parcial de um interpretador!

Segunda Projeção de Futamura

$$\text{target} = [mix](IntL, P)$$

$$\text{compiler} = [mix](mix, IntL)$$

$$\text{target} = [\text{compiler}](P)$$

- Geração de um compilador por meio da auto-aplicação de um avaliador parcial.
- Avaliador parcial tem que ser escrito na mesma linguagem dos programas que processa como entrada!

Terceira Projeção de Futamura

$$\text{compiler} = [\![mix]\!](mix, IntL)$$

$$\text{cogen} = [\![mix]\!](mix, mix)$$

$$\text{compiler} = [\![\text{cogen}]\!](IntL)$$

- Geração de um gerador de compiladores por meio da auto-aplicação de um avaliador parcial!

Auto-Aplicação

- Primeiro avaliador parcial auto-aplicável foi implementado em 1984.
- Autores: Jones, Sestoft e Sondergaard.
- Linguagem Lisp.
- Usado para gerar compiladores simples (*toy examples*).

Conclusões

- Avaliação Parcial pode ser utilizada em muitas áreas para melhorar eficiência de programas.
- A geração de compiladores dirigida por semântica é outra importante aplicação.
- Avaliadores parciais foram desenvolvidos com sucesso para linguagens funcionais, lógicas e imperativas.

Conclusões

- SIMILIX:
  - Importante avaliador parcial para Scheme.
  - Experiências com geração de compiladores (ex: Miranda para Scheme).
- C-MIX:
  - Avaliador parcial para ANSI-C.
  - Desenvolvido pelo DIKU (www.diku.dk/research-groups/topps)
  - Utilizado em diversas aplicações (ex: computação gráfica - *ray tracing*)

Conclusões

- Abordagem ***Cogen***: nova abordagem para avaliação parcial:
  - Consiste em escrever à mão o programa cogen, que pode ser gerado pela Terceira Projeção de Futamura.
  - Utilizada no C-Mix (versões mais recentes) e outros projetos.
  - Permite resultados práticos mais satisfatórios, especialmente na geração de compiladores.