INF05516 - Semântica formal N
Ciência da Computação - UFRGS
2006-2

Marcus Ritt
mrpritt@inf.ufrgs.br

30/08/2006

Introdução 2 / 30

Agenda

Última aula:

- Introdução em OCaml.

Hoje:

- IMP em OCaml. Introdução ao cálculo lambda.

v1951 Semântica formal N, aula 7 – 3 / 30

O cálculo lambda 4 / 30

Introdução

- Peter Landin observou que uma linguagem de programação pode ser compreendida formulando-a em um núcleo pequeno capturando suas características essenciais.
- Ele usou o cálculo lambda como núcleo.
- Outras formas (linguagens ou partes de linguagens) derivadas, podem ser compreendidas traduzindo-as para esse núcleo.

v1951 Semântica formal N, aula 7 – 5 / 30

Porque o cálculo lambda

- Alonzo Church inventou o cálculo lambda para estudar a computabilidade (1936).
- Toda a computação pode ser reduzida às operações básicas de definição de função a aplicação de função.
- Logo, o cálculo lambda é um núcleo bem simples: Isso facilite a analise e o tratamento matemático.
- Mesmo assim, o cálculo lambda é computacionalmente completo.

Aplicações

- O cálculo lambda se aplica na especificação, projeto e implementação de linguagens de programação (vamos usar na semântica denotational!).
- Ele também pode ser vista como uma linguagem de programação simples, e é a base das linguagens funcionais.
- Por exemplo a linguagem LISP (John McCarthy, anos 50) foi implementada baseada no cálculo lambda.

Cálculo lambda

Na forma pura, o cálcula lambda só conhece duas operações:

- A abstração lambda, que cria uma função anônima.
- A aplicação de uma função para um argumento.

Por exemplo a abstração lambda

$$\lambda x.x$$

é a identidade. Tem várias formas extendidos, por exemplo com operadores e constantes

$$\lambda x.x \times x \times x \qquad \lambda x.0$$

Observe a similaridade com OCaml!

A abstração lambda

- Uma abstração lambda nos libera de nomear cada função. Compare:

$$\lambda x.x \qquad f(x) = x$$

- Ela só tem um argumento. Com abstração repetida, podemos construir uma função com multiplos argumentos

$$\lambda x.\lambda y.x \times y$$

- A variável da abstração é entre λ e .: Depois segue o *escopo* dessa variável. O escopo vai até o mais direita possível.
- A função de uma abstração é polimórfico.

A abstração lambda (2)

- O nome da variável não importa:

$$\lambda x.x \qquad \lambda y.y \qquad \lambda z.z$$

  denota a mesma função. Compare com

$$\int f(x)dx \qquad \int f(y)dy \qquad \int f(z)dz$$

A aplicação

- A aplição é notado simplesmente como juxtaposição de uma função e um argumento. Compare $f(x)$ e fx.
- Informalmente, o argumento é substituido pelo variável da função:

$$(\lambda x.x)0 \to 0$$
$$(\lambda x.x \times x \times x)2 \to 2 \times 2 \times 2 \to 8$$
$$(\lambda x.x)(\lambda x.x) \to (\lambda x.x)$$
$$(\lambda x.\lambda y.x \times y)3\,4 \to (\lambda y.3 \times y)4 \to 3 \times 4 \to 12$$

- Observe como no último exemplo a função é aplicada argumento pelo argumento. A aplicação é associativa à esquerda

$$fxy \Leftrightarrow ((fx)y)$$

Sintaxe do cálculo lambda puro

- Precisamos um conjunto (infinito) de variáveis V (com meta-variável $\mathbf{v} \in \mathrm{V}$).
- Expressões em cálculo lambda puro são

$$\mathbf{e} ::= \mathbf{v} | \lambda \mathbf{v}.\mathbf{e} | \mathbf{e_1}\ \mathbf{e_2}.$$

- $\lambda \mathbf{v}.\mathbf{e}$ é uma função anônima com argumento $\mathbf{v}$ e escopo $\mathbf{e}$.
- $\mathbf{e_1}\ \mathbf{e_2}$ é a aplicação de $\mathbf{e_1}$ a $\mathbf{e_2}$
- Exemplos

$$y$$
$$\lambda x.x$$
$$(\lambda x.xx)(\lambda x.xx)$$

Variáveis

- Uma variável $\mathbf{v}$ que occore em uma expressão lambda no escopo de uma abstração $\lambda \mathbf{v}$ é *ligada*. Senão ela é *livre*.
- Podemos definir indutivamente o conjunto $L(e)$ de variáveis livres
  - Se $e \equiv \mathbf{v}$, $L(\mathbf{e}) = \{v\}$
  - Se $e \equiv e_1\, e_2$, $L(e) = L(e_1) \cup L(e_2)$
  - Se $e \equiv \lambda v.e'$, $L(e) = L(e') \setminus \{v\}$
- Uma expressão e sem variáveis livres ($L(e) = \emptyset$) é *fechada*.

Substituição

A substituição de uma varíavel v por uma expressão $\mathbf{e}'$ em uma expressão $\mathbf{e}$ é denotada $\mathbf{e}[\mathbf{e}'/v]$ e é definido indutivamente:

- Se $\mathbf{e} \equiv \mathbf{v}$
  - Se $\mathbf{v} \equiv v$: $\mathbf{e}[\mathbf{e}'/\mathbf{v}] = \mathbf{e}'$
  - Se $\mathbf{v} \not\equiv v$: $\mathbf{e}[\mathbf{e}'/\mathbf{v}] = \mathbf{e}$
- Se $\mathbf{e} \equiv \mathbf{e_1}\,\mathbf{e_2}$: $\mathbf{e}[\mathbf{e}'/v] = \mathbf{e_1}[\mathbf{e}'/v]\,\mathbf{e_2}[\mathbf{e}'/v]$
- Se $\mathbf{e} \equiv \lambda x.\mathbf{e_1}$:
  - Se $x \equiv v$, $\mathbf{e}[\mathbf{e}'/v] = \mathbf{e}$
  - Se $x \not\equiv v$ e $x \notin L(\mathbf{e}')$: $\mathbf{e}[\mathbf{e}'/v] = \lambda x.\mathbf{e_1}[\mathbf{e}'/v]$
  - Se $x \not\equiv v$ e $x \in L(\mathbf{e}')$: $\mathbf{e}[\mathbf{e}'/v] = \lambda z.\mathbf{e_1}[z/x][\mathbf{e}'/v]$ com $z \not\equiv v$ e $z \notin L(\mathbf{e_1}\mathbf{e}')$
- O último substituição renomea a variável de uma abstração. Em geral, consideramos termos do cálculo lambda com variáveis renomeadas equivalente ou *congruentes*-α.

Exemplos

$$\lambda x.\lambda y.x \equiv \lambda z.\lambda y.z \equiv \lambda z.\lambda x.z$$
$$(\lambda x.xy)[y/x] = \lambda x.x$$
$$(\lambda x.xy)[z/y] = \lambda x.xz$$
$$(\lambda x.xy)[x/y] = \lambda z.zy$$

Exemplos: Expressões

$$(\lambda x.x)0$$
$$(\lambda x.x)(\lambda x.x)0$$
$$(\lambda x.\lambda y.xy)(\lambda x.x)0$$

Semântica 18 / 30

SOE do cálculo lambda

$$\frac{\mathbf{e_1} \to \mathbf{e_1'}}{\mathbf{e_1}\,\mathbf{e_2} \to \mathbf{e_1'}\,\mathbf{e_2}}\ \mathsf{app}_1$$

$$\frac{\mathbf{e_2} \to \mathbf{e_2'}}{\mathbf{e_1}\,\mathbf{e_2} \to \mathbf{e_1}\,\mathbf{e_2'}}\ \mathsf{app}_2$$

$$\frac{\mathbf{e} \to \mathbf{e'}}{\lambda \mathbf{v}.\mathbf{e} \to \lambda \mathbf{v}.\mathbf{e'}}\ \mathsf{abs}$$

Uma redução β aplica funções:

$$\frac{}{(\lambda \mathbf{v}.\mathbf{e})\mathbf{e'} \to \mathbf{e}[\mathbf{e'}/\mathbf{v}]}\ \beta$$

$(\lambda \mathbf{v}.\mathbf{e})\mathbf{e'}$ se chama β-redex.

Exemplos

Seja $\mathrm{I} \equiv \lambda x.x$.

$$\underline{(\lambda x.\lambda y.x\, y)\mathrm{I}}\,\mathrm{I} \to_{app_1,\beta} \underline{(\lambda y.\mathrm{I}\, y)\mathrm{I}} \to_\beta \underline{\mathrm{I}\,\mathrm{I}} \to_\beta \mathrm{I}$$

$$\underline{(\lambda x.\lambda y.x\, y)\mathrm{I}}\,\mathrm{I} \to_{app_1,\beta} (\lambda y.\underline{\mathrm{I}y)\mathrm{I}} \to_\beta \underline{(\lambda y.y)\mathrm{I}} \to_\beta \mathrm{I}$$

$$(\lambda x.x\, x)(\lambda x.x\, x) \to_\beta (\lambda x.x\, x)(\lambda x.x\, x)$$

Observações:

- Uma expressão lambda tem possívelmente reduções diferentes.
- Uma redução não necessariamente é finita (termina).
- Questão: Se a redução termina, o resultado final é sempre o mesmo?

Formas normais

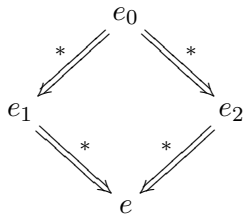
- Em comparação com o cálculo lambda, a semântica operacional de IMP tem um estado e resulta em um estado final.
- O que seria um "valor" adequada de uma expressão lambda?
- Uma expressão que não contém β-redexes é na *formal normal*.
- O exemplos mostram que existem expressões lambda, que não tem uma forma normal.

Confluência

- Mas se existe uma forma normal, ela é única?
- Essa característica se chama *confluência* ou *propriedade de Church-Rosser*:

$$(e_0 \to^* e_1 \wedge e \to^* e_2) \to \exists e(e_1 \to^* e \wedge e_2 \to^* e)$$

- Nosso conjunto de regras (que se chama redução β completa) é confluente.
- Logo, duas formas normais são identicas (módulo congruência-α).
- Compare com a noção de determinísmo na semântica operacional!



Variação 1

Suponha que usamos só as regras app_1 e β.

- Agora não podemos mais simplificar o lado direito de uma aplicação, nem o escopo de uma abstração.
- Em outras palavras, não avaliamos os argumentos antes da aplicação!
- Obtemos uma semântica chamado *call-by-name* ou com *avaliação preguiçosa* (inglês: lazy evaluation)!

Variação 2

Considere a seguinte modificação:

- Seja val a categoria sintatica de *valores*

$$\mathbf{c} ::= \lambda \mathbf{v}.\mathbf{e}$$

  com $\mathbf{c} \in \mathrm{val}$.
- Usa as regras originais sem "abs" e modifique as regras "app_2" e "β":

$$\frac{\mathbf{e_2} \to \mathbf{e_2'}}{\mathbf{c_1}\,\mathbf{e_2} \to \mathbf{c_1}\,\mathbf{e_2'}}\,\mathsf{app}_2$$

$$\frac{}{(\lambda \mathbf{v}.\mathbf{e})\,\mathbf{v'} \to \mathbf{e}[\mathbf{v'}/\mathbf{v}]}\,\beta$$

- Uma abstração se aplica só para valores.
- Obtemos uma semântica de *call-by-value* ou com *avaliação gulosa* (inglês: strict evaluation)!

Exemplos

Sistema completo:

$$\underline{(\lambda x.\lambda y.\mathrm{I}y)\Omega} \to_\beta \lambda y.\underline{\mathrm{I}y} \to_{abs,\beta} \lambda y.y \equiv \mathrm{I}$$

Sistema preguiçoso:

$$\underline{(\lambda x.\lambda y.\mathrm{I}y)\Omega} \to_\beta \lambda y.\mathrm{I}y \not\to$$

Sistema guloso:

$$(\lambda x.\lambda y.\mathrm{I}y)\Omega \not\to$$

Tese de Church-Turing

As funções efetivamente computáveis sobre os números inteiros positivos são precisamente as funções que podem ser definidas na cálculo lambda puro (e computáveis por máquinas de Turing).

Poder do cálculo lambda puro

- Como é possível que cálculo lambda é tão poderoso?
- Varios elementos de linguagens de programação tem uma codificação no cálculo lambda, por exemplo,
  - Valores de verdade, funções booleanas e um condicional
  - Números inteiros
  - Listas

Exemplo: Valores de verdade

- $\text{true} \equiv \lambda t.\lambda f.t$
- $\text{false} \equiv \lambda t.\lambda f.f$
- $\text{and} \equiv \lambda v.\lambda v'.v\ v'\ \text{false}$
- $\text{or} \equiv \lambda v.\lambda v'.v\ \text{true}\ v'$

Exemplo:

$$\underline{\text{and true}}\,\text{true} \to_{app_1,\beta} \underline{(\lambda v'.\text{true}\ v'\ \text{false})\text{true}}$$
$$\to_\beta \text{true true false} \to^* \text{true}$$

Exemplo: Números inteiros

- $0 \equiv \lambda s.\lambda z.z$
- $1 \equiv \lambda s.\lambda z.s\ z$
- $2 \equiv \lambda s.\lambda z.s\ (s\ z)$
- $3 \equiv \lambda s.\lambda z.s\ (s\ (s\ z))$
- etc. $0, 1, 2, \ldots$ se chamam *numerais de Church*.
- $\text{succ} = \lambda n.\lambda s.\lambda z.s(nsz)$
- $\text{plus} = \lambda m.\lambda n.\lambda s.\lambda z.ms(nsz)$
- $\text{times} = \lambda m.\lambda n.m(\text{plus}\ n)0$

Exemplos:

$$\underline{\text{succ}\ 1} \to_\beta \lambda s.\lambda z.s(\underline{1\ s\ z}) \to_{abs,abs,\beta} \lambda s.\lambda z.s(s\ z) \equiv 2$$
$$\underline{\text{plus}\ 1}\ 2 \to^* \lambda s.\lambda z.\underline{1\ s\ (2\ s\ z)} \to^* \lambda s.\lambda z.s\ (\underline{2\ s\ z})$$
$$\to_{abs,abs,app_2,\beta} \lambda s.\lambda z.s\ (s\ (s\ z)) \equiv 3$$