Introdução à Teoria Formal de Objetos

Pedro Bruel, António Miranda

09 de Junho

Programação Orientada a Objetos 2015

Índice

Contextualização

Cálculo-λ

- É um sistema formal para expressão de cálculos;
- Representa a Aplicação e Abstração de funções;
- Pode ser Tipado ou Não-Tipado;

Cálculo-λ

- É um sistema formal para expressão de cálculos;
- Representa a Aplicação e Abstração de funções;
- Pode ser Tipado ou Não-Tipado;
- Aplicação ao ”Problema de Decisão” (*Entscheidungsproblem*);
- Implementação de Linguagens Funcionais.

Cálculo-λ

Function calculi and their features

Calculus:	λ	$\mathbf{F_1}$	$\mathbf{F_{1<:}}$	$\mathbf{F_{1\mu}}$	$\mathbf{F_{1<:\mu}}$	**impλ**	**F**	$\mathbf{F_{<:}}$	$\mathbf{F_{\mu}}$	$\mathbf{F_{<:\mu}}$
Defined in:	6.3	7.3	8.1	9.1	9.2	10.3	13.1–13.2	13.1–13.2	13.1–13.2	13.1–13.2
functions	•	•	•	•	•	•	•	•	•	•
function types		•	•	•	•		•	•	•	•
subtyping			•		•			•		•
recursive types				•	•				•	•
side-effects						•				
quantified types							•	•	•	•

Cálculo-λ: Definição

Expressões são compostas de:

- $v_1, v_2, \ldots, v_n$
- Abstração: λ e .
- Aplicação: ()

Cálculo-λ: Definição

Expressões são compostas de:

- $v_1, v_2, \ldots, v_n$
- Abstração: λ e .
- Aplicação: ()

Conjunto Λ de Expressões:

- $x \in \Lambda$
- $x \in \Lambda, M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$
- $M, N \in \Lambda \Rightarrow (M, N) \in \Lambda$

Cálculo-λ: Definição

Variáveis Livres:

- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) \backslash \{x\}$
- $FV(MN) = FV(M) \bigcup FV(N)$

Cálculo-λ: Definição

Variáveis Livres:

- $FV(x) = \{x\}$
- $FV(\lambda x.M) = FV(M) \backslash \{x\}$
- $FV(MN) = FV(M) \bigcup FV(N)$

Representação em Lisp, Haskel, ...:

```
(lambda (x) (* x x))
```

Tipos e Categorias

Teoria de Tipos

Principia Mathematica:

- Problema: Paradoxo de Russel;

Teoria de Tipos

Principia Mathematica:

- Problema: Paradoxo de Russel;
- Solução: Restringir operações a certos Tipos.

Teoria de Tipos

Principia Mathematica:

- Problema: Paradoxo de Russel;
- Solução: Restringir operações a certos Tipos.

Exemplo em julialang:

```
(Integer       <: Number)         => true
(FloatingPoint <: Number)         => true
(Int64         <: Integer)        => true
(Float64       <: FloatingPoint) => true
(ASCIIString   <: Number)         => false
```

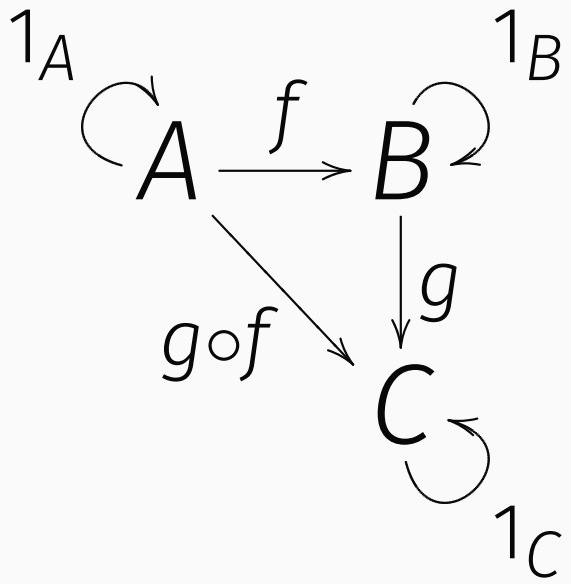
Teoria de Categorias

Categorias:

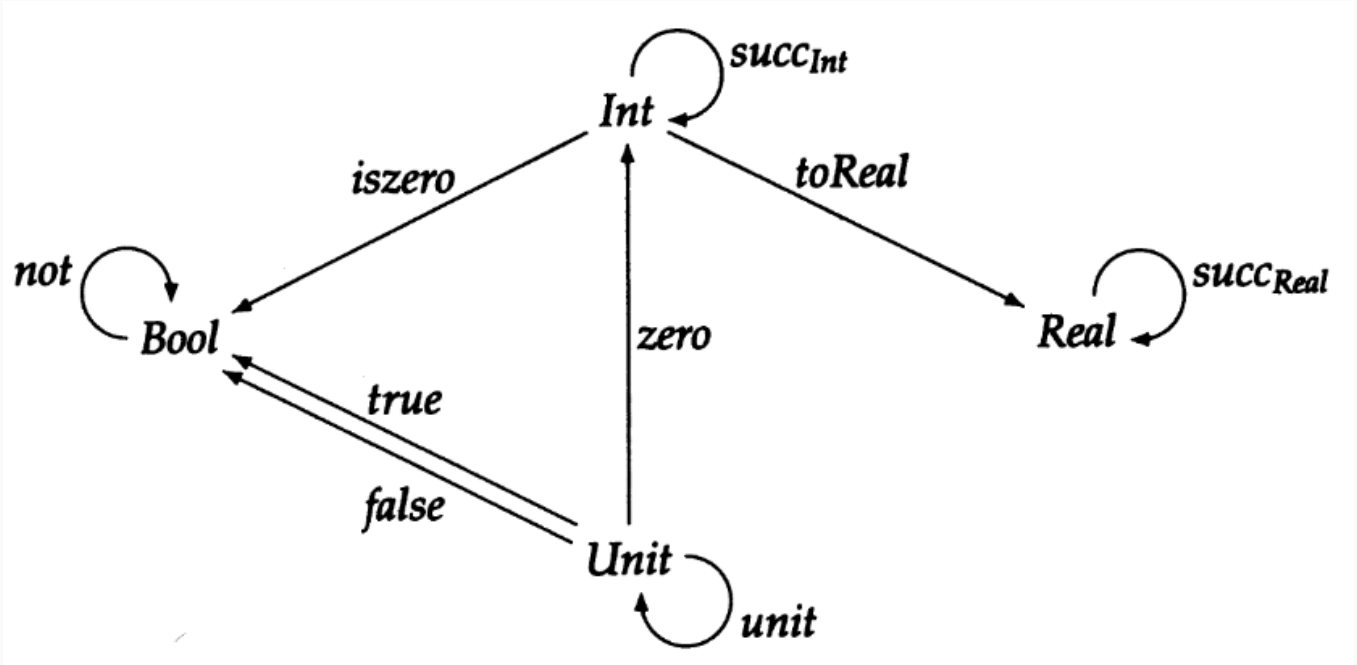
- Estruturas compostas de *objetos* e *morfismos*;
- Analogia com Sistemas de Tipos.

Categorias:

- Estruturas compostas de *objetos* e *morfismos*;
- Analogia com Sistemas de Tipos.



Sistema de Tipos como Categoria:



Exemplo

Covariância e Contravariância:

- Propriedades de *Functores* (Mapeamentos entre Categorias);
- Contravariância: Mapeamento do oposto de uma Categoria.

Exemplo

Covariância e Contravariância:

- Propriedades de *Functores* (Mapeamentos entre Categorias);
- Contravariância: Mapeamento do oposto de uma Categoria.

Em Linguagens Orientadas a Objetos:

- Restrição de operações a certos tipos e subtipos;
- Relações entre tipos;

Exemplo: Covariância

Sejam A e B tipos, então a composição AxB é covariante nos dois tipos, pois:

$AxB <: A'xB' \Leftrightarrow A <: A', B <: B'$

Exemplo: Covariância

Sejam A e B tipos, então a composição AxB é covariante nos dois tipos, pois:

$AxB <: A'xB' \Leftrightarrow A <: A', B <: B'$

Exemplo em julialang:

```
typeof((2::Int64, 2.0::Float64)) <: (Number, Number)
true
typeof((2::Int64, 2.0::Float64)) <: (Integer, FloatingPoint)
true
typeof((2::Int64, 2.0::Float64)) <: (Integer, Integer)
false
```

Exemplo: Contravariância

Sejam A e B tipos, então a função $f : A \to B$ é covariante em B, mas contravariante em A, pois:

$f : A \to B <: f : A' \to B' \Leftrightarrow A' <: A, B <: B'$

Exemplo: Contravariância

Sejam A e B tipos, então a função $f : A \to B$ é covariante em B, mas contravariante em A, pois:

$f : A \to B <: f : A' \to B' \Leftrightarrow A' <: A, B <: B'$

Exemplo em julialang:

```
function f{T <: Integer}(n::T)
   return string(n)
end
```

Object Calculi

Object calculi and their features

Calculus:	ς	$\mathbf{Ob_1}$	$\mathbf{Ob_{1<:}}$	*nn*	$\mathbf{Ob_{1\mu}}$	$\mathbf{Ob_{1<:\mu}}$	*nn*	$\mathbf{imp\varsigma}$	*nn*
Defined in:	6.2	7.3	8.1	8.7	9.1	9.2	9.7	10.1	11.1
objects	•	•	•	•	•	•	•	•	•
object types		•	•	•	•	•	•		•
subtyping			•	•		•	•		•
variance				•					
recursive types					•	•	•		
dynamic types							•		
side-effects								•	•

Calculus:	**Ob**	$\mathbf{Ob_\mu}$	$\mathbf{Ob_{<:}}$	$\mathbf{Ob_{<:\mu}}$	$\mathbf{\varsigma Ob}$	**S**	$\mathbf{S_\forall}$	*nn*	$Ob_{\omega<:\mu}$
Defined in:	13.1–13.2	13.1–13.2	13.1–13.2	13.1–13.2	13.5	16.2	16.3	17.1–17.3	20.1–20.3
objects	•	•	•	•	•	•	•	•	•
object types	•	•	•	•	•	•	•	•	•
subtyping			•	•	•	•	•	•	•
variance			◦	◦		•	•	•	•
recursive types		•		•					•
dynamic types									
side-effects								•	
quantified types	•	•	•	•			•	•	•
Self types				◦	•	•	•	•	◦
structural rules						•	•	•	•
type operators									•

Cálculo-ς

Cálculo-ς

Para formalizar a ideia de um objeto, no livro *A theory of objects* de Martín Abadi e Luca Cardelli, foi desenvolvido o cálculo-ς (sigma).

Baseado no já conhecido cálculo-λ (lambda), trata-se de um cálculo não-tipado que define a semântica para os objetos e trata-os como primitivas.

Cálculo-ς: Primitiva Objeto

No cálculo-ς, um objeto é considerado uma estrutura computacional. É uma coleção de atributos (não anônimos), onde todos são métodos.

Um método é composto por uma variável que representa o *self* e por um corpo que produz o resultado.

As únicas operações válidas sobre objetos são atualização e invocação de métodos.

Cálculo-ς: Primitiva Objeto

A notação usada para um objeto e seus métodos:

- ς(x)b - método com x como parâmetro *self* e corpo b;
- $[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ - um objeto com n métodos, onde cada método é identificado pelo $l_1, \ldots, l_n$;
- o.l - o objeto o invoca o método l;
- $o.l \overleftarrow{\leftarrow} \varsigma(x)b$ - atualizando o método l do objeto o com o método ς(x)b (a atualização de um método produz uma cópia do objeto o com o ς(x)b no lugar do método l).

Exemplo de um objeto: $[l_1 = \varsigma(x_1)[], l_2 = \varsigma(x_2)x_2.l_1]$

Cálculo-ς: Primitiva Objeto

Métodos que não usam o parâmetro *self*, $l_1 = \varsigma(x_1)[]$ do exemplo anterior, são considerados campos do objeto. Simplificações na notação de um campo:

- $[\ldots, l = b, \ldots]$ é equivalente a $[\ldots, l = \varsigma(y)b, \ldots]$ para um parâmetro y (*self*) não usado (declaração de um campo);
- o.l := b é equivalente a o.l $\overleftarrow{=}$ ς(y)b para um y não usado (atualização de um campo).

Terminologia para atributos e operações num objeto:

		Atributos	
		campos	métodos
Operações	seleção	seleção	invocação
	atualização	atualização	atualização

Cálculo-ς: Semântica das primitivas

A execução de um termo num cálculo-ς pode ser expresso como uma sequência de redução de passos. Isso é conhecido por **semântica primitiva**. Ela representa da forma mais simples e direta possível a semântica pretendida do objeto.

Para definir essas semânticas primitivas, foram introduzidas as seguintes notações:

- $\Phi_i{}^{i \in 1 \ldots n}$ denota $\Phi_1, \Phi_2, \ldots, \Phi_n$;
- b $\mapsto$ c significa que b reduz a c em um passo;
- b $\{\{x \leftarrow c\}\}$ indica a substituição do termo c para as ocorrências de x em b.

Cálculo-ς: Semântica das primitivas

Passos da redução para as operações existentes no cálculo-ς:
Seja $o \equiv [l_i = \varsigma(x_i)b_i \ ^{i \in 1 \ldots n}]$ um objeto, onde todos os l_i são distintos.
Temos,

- $o.l_j \mapsto b_j \{\{x_j \leftarrow o\}\}$ $(j \in 1 \ldots n)$ - redução de uma invocação;
- $o.l \overset{\leftarrow}{\leftarrow} \varsigma(y)b \mapsto [l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i \ ^{i \in 1 \ldots n}]$ $(j \in 1 \ldots n)$ - redução de uma atualização.

A invocação do $o.l_j$ consiste na substituição do objeto *host* pelo *self* no copro do método l_j.
A atualização do método $o.l \overset{\leftarrow}{\leftarrow} \varsigma(y)b$ se reduz a uma cópia do objeto *host* com o método trocado pela versão atualizada.

Cálculo-ς: Semântica das primitivas

Supondo que os símbolos $\doteq$ e $\equiv$ significam respectivamente "igual por definição" e "sintaticamente igual", vamos apontar alguns exemplos de reduções.

A substituição própria (*self-substitution*) está no *core* das semânticas primitivas da invocação de um método, portanto, é fácil programar computações não-terminais sem o uso explícito da recursão.

Seja
$o_2 \doteq [\mathrm{l} = \varsigma(\mathrm{x})\mathrm{x}.\mathrm{l}]$
então
$o_2.\mathrm{l} \mapsto \mathrm{x}.\mathrm{l}\{\{\mathrm{x} \leftarrow o_2\}\} \equiv o_2.\mathrm{l} \mapsto \ldots$

Cálculo-ς: Semântica das primitivas

A substituição própria também permite retornar e modificar o *self*.

Seja
$o_3 \doteq [l = \varsigma(x)x]$
então
$o_3.l \mapsto x\{\{x \leftarrow o_3\}\} \equiv o_3$

Seja
$o_4 \doteq [l = \varsigma(y)(y.l \overset{\leftarrow}{\leftarrow} \varsigma(x)x)]$
então
$o_4.l \mapsto (y.l \overset{\leftarrow}{\leftarrow} \varsigma(x)x)\{\{y \leftarrow o_4\}\} \equiv o_3$

OBS: Em linguagens baseadas em classes, é comum um método modificar seus atributos, apesar desses atributos serem campos e não métodos.

Cálculo-ς: Sintaxe

A sintaxe do cálculo-ς é descrita pela gramática baixo:

a,b ::= (termo)
- x (variável)
- $[l_i = \varsigma(x_i)b_i$ (i $\in 1 \ldots$ n)] (criação de um objeto, l_i distintos)
- a.l (seleção de campos ou invocação de métodos)
- a.l $\overleftarrow{=}$ ς(x)b (atualização de campos ou métodos)

Qualquer expressão gerada por essa gramática é um termo-ς.

A notação a,b ::= ... descreve indutivamente um conjunto de expressões, onde a e b pertencem ao conjunto das expressões a serem definidos.

Cálculo-ς: Sintaxe

As outras definições significam que um termo a ou b podem ser uma:

- variável x;
- expressão da forma $[l_i = \varsigma(x_i)b_i \ ^{i \in 1 \ldots n}]$ (onde l_i é um rótulo, x_i é uma variável e b_i é um termo);
- expressão da forma a.l (onde a é um termo e l é o rótulo de um método ou campo);
- expressão da forma a.l $\overleftarrow{=} \varsigma$(x)b (onde a e b são termos, l é o rótulo de um método ou campo e x uma variável).

Cálculo-ς: Exemplo da calculadora

calculadora $\doteq$

[arg = 0.0,
aux = 0.0,
inserir = $\varsigma(s)\lambda(n)s.arg := n$,
mais = $\varsigma(s)(s.aux := s.igual).igual \leftarrow \varsigma(s')s'.aux + s'.arg$,
menos = $\varsigma(s)(s.aux := s.igual).igual \leftarrow \varsigma(s')s'.aux - s'.arg$,
igual = $\varsigma(s)s.arg$,
reiniciar = $\varsigma(s)(s.arg := 0.0).aux := 0.0$]

As variáveis arg e aux são usadas nas operações internas da calculadora.
Os métodos inserir, mais, menos, igual e reiniciar provêm a interface.

Cálculo-ς: Exemplo da calculadora

Exemplo de uso e comportamento da calculadora.

calculadora.inserir(5.0).igual = 5.0
calculadora.inserir(5.0).menos.inserir(3.5).igual = 1.5
calculadora.inserir(5.0).mais.mais.igual = 15.0
calculadora.inserir(5.0).reiniciar = 0.0

Cálculo-ς: Classe

No cálculo-ς, uma classe é considera um gerador de objetos com informações sobre si mesma. Em outras palavras podemos dizer que uma classe é uma coleção de pré-métodos (informações) que possui um método *new* para gerar os novos objetos.

Um pré-método é um método da forma:

$$l_i = \varsigma(z)\ \lambda(x_i)b_i \text{ onde i} \in 1 \ldots \text{n}$$

ou

$$l_i = \lambda(x_i)b_i \text{ onde i} \in 1 \ldots \text{n}$$

OBS: Informações de uma classe (pré-métodos) podem ser contribuições de outras classes.

Cálculo-ς: Classe

$$
\begin{aligned}
c \doteq & \\
& [\quad \text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1\ldots n}], \\
& \quad\ l_i = \lambda(s)b_i\ ^{i\in 1\ldots n}\ ]
\end{aligned}
$$

O método *new* simplesmente aplica o *self* do novo objeto na coleção de pré-métodos da classe, que por conseguinte os transforma em métodos.

Dada uma classe c, a invocação c.new um objeto do tipo c. Exemplo:

$$o \doteq c.\text{new} = [l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]$$

Cálculo-ς: Herança

Consiste em reusar pré-métodos entre classes (ou *traits*). Exemplo, se uma classe c' é definida reusando todos os pré-métodos de uma classe c (c.l_j $j \in 1 \ldots n$) e adicionando mais pré-métodos (λ(s)b_k $^{k \in n+1 \ldots n+m}$), informalmente podemos dizer que c' é uma subclasse de c.

$$
\begin{aligned}
c' \doteq \\
\quad [\quad & \text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1 \ldots n+m}], \\
& l_j = \text{c}.l_j \ ^{j \in 1 \ldots n}, \\
& l_k = \lambda(s)b_k \ ^{k \in n+1 \ldots n+m} \ ]
\end{aligned}
$$

Cálculo-ς: Herança

Uma subclasse pode sobrescrever pré-métodos em vez de herdá-los. Exemplo, se uma classe c'' é definida reusando os primeiros n-1 pré-métodos de uma classe c e sobrecarregando o último pré-método.

$$
\begin{aligned}
&c'' \doteq \\
&\quad [\quad \text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i\in 1\ldots n}], \\
&\qquad l_j = c.l_j{}^{\,j\in 1\ldots n-1}, \\
&\qquad l_k = \lambda(s)\ldots c.l_n(s)\ldots c.l_p(s)\ldots]
\end{aligned}
$$

O método sobrecarregado l_n pode referenciar a um pré-método original de c como $c.l_n$, ou até a um outro pré-método de uma super classe de c, com $c.l_p$, onde $p \in 1\ldots n$. Assim modelamos o comportamento do **super**.
OBS: A herança múltipla é obtida através do reúso de pré-métodos de múltiplas classes (ou *traits*).

Referências

Referências

1. Abadi, M., e Cardelli, L. (1996). *A* theory of objects. Springer Science & Business Media.
2. Pierce, B. (2002). *T*ypes and programming Languages. The MIT Press.

Obrigado! Perguntas?