Uma Introdução à Teoria de Tipos

Flávio L. C. de Moura*

21 de Agosto de 2017

Iniciaremos este trabalho estabelecendo uma relação direta entre lógica e computação. Esta relação é conhecida como o *isomorfismo de Curry-Howard*[1]. Considere inicialmente o fragmento implicacional da lógica proposicional intuicionista. Neste fragmento, fórmulas são construídas a partir da seguinte gramática:

$$\varphi ::= V \mid (\varphi \to \varphi) \tag{1}$$

onde V denota um conjunto (enumerável) de variáveis proposicionais, $(\varphi \to \varphi)$ denota como uma nova fórmula pode ser construída a partir de duas fórmulas dadas. Um sistema de dedução natural associado a este fragmento possui as seguintes regras, onde Δ denota um conjunto (finito) de fórmulas proposicionais construídas a partir de (1):

$$\frac{}{\Delta \cup \{\tau\} \vdash \tau}\ (\text{Ax})$$

$$\frac{\Delta \vdash \sigma \to \tau \qquad \Delta \vdash \sigma}{\Delta \vdash \tau}\ (\to_e)$$

$$\frac{\Delta, \sigma \vdash \tau}{\Delta \vdash \sigma \to \tau}\ (\to_i)$$

A regra (Ax) é um axioma que nos permite deduzir qualquer informação existente no contexto. A regra $(\to_e)$ é conhecida como *modus ponens*, e nos permite provar τ a partir de uma implicação $\sigma \to \tau$ e do antecedente σ. A terceira regra nos diz como podemos construir uma implicação $\sigma \to \tau$, a partir de uma prova de τ que assuma σ como hipótese.

Por volta de 1930, Haskell Brooks Curry observou que uma fórmula implicacional $A \to B$ corresponde ao tipo das funções de domínio A e contradomínio B, e que inferir B a partir de $A \to B$ e A corresponde a aplicar a hipótese $A \to B$ à hipótese A, ou seja, aplicar a função de tipo $A \to B$ ao argumento de tipo A. Analogamente, provar uma implicação $A \to B$ corresponde a inferir B a partir da suposição A, o que corresponde a construir uma função que leva elementos de tipo A (domínio) em elementos de tipo B (contradomínio). Em 1969, um trabalho de William Howard mostrou que esta correspondência não é uma mera coincidência, mas um princípio fundamental.

A seguir, se φ representa uma fórmula, e t uma função, então escreveremos $t : \varphi$ para dizer que a função (ou o termo) t tem tipo φ. Por exemplo, a função sucessor `S` sobre os números naturais tem tipo $\mathbb{N} \to \mathbb{N}$, isto é, `S` é uma função que recebe um natural como argumento e retorna um natural como resultado. Podemos representar este fato escrevendo $\mathtt{S} : \mathbb{N} \to \mathbb{N}$. Analogamente, podemos expressar o fato de que o número 0 é um natural escrevendo $0 : \mathbb{N}$. Agora podemos aplicar a função sucessor a 0, obtendo a seguinte derivação:

$$\frac{\mathtt{S} : \mathbb{N} \to \mathbb{N} \qquad 0 : \mathbb{N}}{(\mathtt{S}\ 0) : \mathbb{N}}$$

*contato@flaviomoura.mat.br

[1] *Proof as Terms or Propositions as Types (PAT)*

A conclusão é que $(\mathtt{S}\ 0)$ é um número natural. Podemos utilizar a derivação acima para mostrar que $(\mathtt{S}(\mathtt{S}\ 0)) : \mathbb{N}$:

$$\dfrac{\dfrac{\mathtt{S} : \mathbb{N} \to \mathbb{N} \qquad 0 : \mathbb{N}}{(\mathtt{S}\ 0) : \mathbb{N}} \qquad \mathtt{S} : \mathbb{N} \to \mathbb{N}}{(\mathtt{S}(\mathtt{S}\ 0)) : \mathbb{N}}$$

Estas derivações têm uma estrutura de árvore cuja raiz é o que queremos provar, e cujas folhas correspondem a informações dadas, isto é, hipóteses. Note que as hipóteses na derivação acima correspondem às informações dos tipos das constantes $\mathtt{S}$ e 0. Nas derivações a seguir, utilizaremos variáveis (além das constantes) na construção dos termos, e as informações de tipos para variáveis e constantes serão armazenadas em um conjunto especial que chamaremos de **contexto**. Assim, considerando o contexto $\Gamma = \{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\}$, escreveremos $\Gamma \vdash (\mathtt{S}(\mathtt{S}\ 0)) : \mathbb{N}$ pare denotar que existe uma derivação de $(\mathtt{S}(\mathtt{S}\ 0)) : \mathbb{N}$ a partir de Γ. Em geral, um par da forma $\Gamma \vdash t : \sigma$ é chamado de **sequente**, e representa o fato de que é possível derivar que o termo t tem tipo σ a partir das informações dadas no contexto Γ. Para facilitar a leitura de quais informações estão sendo assumidas como hipóteses em uma derivação, utilizaremos sequentes em cada passo da árvore de derivação. Desta forma, a derivação acima pode ser reescrita da seguinte forma:

$$\dfrac{\dfrac{\{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\} \vdash \mathtt{S} : \mathbb{N} \to \mathbb{N} \qquad \{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\} \vdash 0 : \mathbb{N}}{\{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\} \vdash (\mathtt{S}\ 0) : \mathbb{N}} \qquad \{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\} \vdash \mathtt{S} : \mathbb{N} \to \mathbb{N}}{\{\mathtt{S} : \mathbb{N} \to \mathbb{N}, 0 : \mathbb{N}\} \vdash (\mathtt{S}(\mathtt{S}\ 0)) : \mathbb{N}}$$

Veja que as folhas desta árvore de derivação diz simplesmente que uma informação do contexto pode ser extraída gratuitamente.

Considerando agora uma teoria de funções mais geral, utilizaremos a notação $\lambda_{x:\sigma}.t$ para denotar a função (anônima) que tem a variável x, do tipo σ, como parâmetro, e corpo t. Por exemplo, a função que recebe um natural como argumento e retorna o seu sucessor pode ser representada por $(\lambda_{x:\mathbb{N}}.\mathtt{S}\ x)$. Formalmente, a teoria geral de funções com a qual trabalharemos é conhecida como *cálculo* λ.

1 O Cálculo λ

O cálculo λ surgiu no início do século XX com os trabalhos de Alonzo Church [3] e [4]. Esses estudos faziam parte de uma teoria mais geral de funções e lógica de ordem superior, cujo intuito era o de servir como fundamento para a Matemática. Quando Kleene e Rosser [6], então alunos de Church, mostraram a inconsistência dessa teoria, Church abandonou o programa de fundamentos e extraiu a subteoria referente à parte funcional que hoje corresponde ao que chamamos de cálculo λ, cuja consistência foi mostrada por [5].

O cálculo λ é o primeiro sistema de reescrita conhecido no contexto computacional. É também um modelo com uma notação compacta, conveniente para representar funções computáveis, sendo por esse motivo a base do paradigma de programação funcional.

O cálculo λ é uma teoria que modela funções e seu comportamento aplicativo, onde a *aplicação* é uma operação básica: dada uma função f e um argumento a, denotamos o resultado da aplicação da função f ao argumento a por

$$f\ a$$

O cálculo λ possui uma outra operação básica, chamada de *abstração*, que nos permite construir funções. O significado da abstração pode ser compreendido da seguinte forma: se $g(x)$ representa uma expressão, possivelmente contendo a variável x, então representamos por $\lambda_x.g(x)$ como sendo a função que associa à cada argumento a, o valor $g(a)$.

Em [2], Barendregt explica o aparecimento do símbolo λ para denotar a abstração de função: "Em *Principia Mathematica* [7], a notação para função f com $f(x) = 2x + 1$ é $2\widehat{x} + 1$. Church originalmente pretendia utilizar a notação $\widehat{x}.2x+1$. No entanto, o tipógrafo não conseguiu posicionar o símbolo $\widehat{\ }$ em cima da letra x, e o colocou em frente da mesma resultando em $\widehat{\ }x.2x + 1$. Depois outro tipógrafo mudou $\widehat{\ }x.2x + 1$ para $\lambda_x.2x + 1$".

O conjunto dos λ-termos, denotado por Λ, corresponde ao menor conjunto que pode ser construído a partir de um conjunto enumerável de variáveis utilizando-se as operações de aplicação e substituição. Na forma de gramática em BNF esta afirmação corresponde a seguinte construção:

$$\Lambda ::= x \mid (\Lambda\ \Lambda) \mid (\lambda_x.\Lambda)$$

Dada uma função qualquer

$$\lambda_x.M \tag{2}$$

dizemos que x é o seu parâmetro, enquanto que M é o seu *corpo*. Na função (2), as ocorrências de x em M são ditas *ligadas* (pelo abstrator λ_x). A ocorrência de uma variável que não é ligada chama-se *livre*. Por exemplo, em $\lambda_x.(\lambda_y.M)\ N$ o corpo do abstrator λ_y é M enquanto que o corpo do abstrator λ_x é $\lambda_y.M$. Os parênteses podem ser eliminados quando a abrangência dos abstratores é evidente como em $\lambda_x.M$.

Notação.

1. $M\ N_1 \ldots N_k$ é o mesmo que $(\ldots((M\ N_1)N_2)\ldots N_k)$.
2. $\lambda_{x_1 \ldots x_k}.M$ significa $(\lambda_{x_1}.(\lambda_{x_2}.(\ldots(\lambda_{x_k}.M)\ldots)))$.
3. Aplicações têm maior prioridade que abstrações, ou seja $\lambda_x.M\ N = \lambda_x.(M\ N)$.

A noção de *variável livre* e *variável ligada* pode ser formalizada como a seguir:

Definição 1 *O conjunto das variáveis livres do λ-termo M, denotado por $FV(M)$, é definido indutivamente por:*

1. *$FV(x) = \{x\}$, para qualquer variável x;*
2. *$FV(M\ N) = FV(M) \cup FV(N)$;*
3. *$FV(\lambda_x.M) = FV(M) \backslash \{x\}$.*

Note que uma variável pode ocorrer tanto livre quanto ligada em um mesmo λ-termo. De fato, no λ-termo $x\ (\lambda_x.x)$ a primeira ocorrência da variável a é livre enquanto que a segunda ocorrência é ligada. Dizemos que a variável x ocorre livre no λ-termo M quando **todas** as ocorrências de x em M são livres.

Definição 2 *Um λ-termo é dito fechado se não contém variáveis livres.*

Funções são aplicadas a argumentos para gerar resultados. A operação de aplicar uma função $\lambda_x.M$ a um argumento N nos permite definir a noção de computação no cálculo λ por meio de uma regra conhecida como regra (β):

$$(\beta)\quad (\lambda_x.M)\ N \mapsto M\{N/x\} \tag{3}$$

O λ-termo $M\{N/x\}$ representa a substituição por N de todas as ocorrências livres da variável x no λ-termo M. A regra β é a principal regra do cálculo λ, já que esta nos diz como computar o resultado de se aplicar uma função a um argumento. As substituições desempenham um papel fundamental neste contexto. De fato, algumas sutilezas que não são óbvias a partir da definição da regra β devem ser observadas:

- Variáveis livres e ligadas são obviamente objetos distintos em um λ-termo de forma que se durante o processo de computação, ou seja aplicações da regra β, variáveis livres não podem se tornar ligadas.

Por exemplo, $(\lambda_y.x)\{y\ y/x\} \neq \lambda_y.y\ y$. Denotamos o fecho contextual da redução (3) por $\rightarrow_\beta$, isto é:

$$\frac{}{(\lambda_x.M)\ N \rightarrow_\beta M\{N/x\}}\ (\beta) \qquad \frac{M \rightarrow_\beta N}{\lambda_x.M \rightarrow_\beta \lambda_x.N}\ (\xi)$$

$$\frac{M \rightarrow_\beta N}{M\ P \rightarrow_\beta N\ P} \qquad \frac{M \rightarrow_\beta N}{P\ M \rightarrow_\beta P\ N}$$

Adicionalmente, definimos $\twoheadrightarrow_\beta$ como sendo o fecho transitivo-reflexivo de $\to_\beta$.

Assim, sempre que necessário podemos fazer um *renomeamento de variáveis ligadas* de forma que $(\lambda_y.x)\{y\ y/x\} \to_\beta \lambda_z.y\ y$. O renomeamento de variáveis ligadas é denominado *α-conversão.* Termos α-equivalentes são λ-termos iguais a menos dos nomes das variáveis ligadas. Os renomeamentos e as substituições descritos acima não possuem uma descrição formal na linguagem do cálculo λ, e na verdade constituem meta-operações.

A operação de substituição que aparece na definição da regra β é definida como a seguir:

Definição 3 (Substituição) *O resultado de se substituir x por N em M, denotado por $M\{N/x\}$, é definido recursivamente por:*

- $x\{N/x\} = N$.
- $y\{N/x\} = y$, *onde assumimos que* $x \neq y$.
- $(M_1\ M_2)\{N/x\} = M_1\{N/x\}\ M_2\{N/x\}$.
- $(\lambda_x.M)\{N/x\} = \lambda_x.M$.
- $(\lambda_y.M)\{N/x\} = \lambda_y.M\{N/x\}$, *se* $y \notin FV(N)$ *ou* $x \notin FV(M)$.
- $(\lambda_y.M)\{N/x\} = \lambda_z.M\{z/y\}\{N/x\}$, *se* $y \in FV(N)$ *e* $x \in FV(M)$; *onde* $z \notin FV(M)$.

Definição 4 (β-conversão) *A β-conversão é definida como uma sequência de β-reduções ou β-reduções invertidas. A β-conversão entre os termos P e Q é denotada por $P =_\beta Q$.*

Uma propriedade importante das substituições é dada pelo seguinte lema:

Lema 1 (Lema da Substituição) *Sejam $M, N, L \in \Gamma$. Suponha que $x \neq y$ e $x \notin FV(L)$. Então:*

$$M\{N/x\}\{L/y\} =_\beta M\{L/y\}\{N\{L/y\}/x\}$$

Prova. Indução na estrutura de M (exercício).

O cálculo λ é um formalismo expressivo o suficiente para representar qualquer função computável, i.e. o cálculo λ é *Turing-completo.* A prova clássica deste fato é feita mostrando que qualquer função recursiva pode ser representada por um λ-termo (veja por exemplo [1]). Em particular, o cálculo λ é capaz de representar processos computacionais infinitos. De fato, o exemplo clássico de redução não-terminante é dado por:

$$(\lambda_x.x\ x)(\lambda_x.x\ x) \to_\beta (\lambda_x.x\ x)(\lambda_x.x\ x) \to_\beta \dots$$

Os teoremas de ponto fixo desempenham um papel importante em Matemática e Computação (ver capítulo 4 de [1]). O teorema do ponto fixo no cálculo λ é dado por:

Teorema 1 *Para todo $F \in \Lambda$ existe $X \in \Lambda$ tal que $F\ X =_\beta X$.*
Prova. Considere $W = \lambda_u.F(u\ u)$. Então faça $X = W\ W$. Neste caso, temos $X = (\lambda_u.F(u\ u))W =_\beta F(W\ W) =_\beta F\ X$.

Uma das propriedades mais importantes do cálculo λ é conhecida como *confluência.* Intuitivamente, dizemos que um sistema de reescrita é confluente se qualquer divergência gerada pode ser juntada. Formalmente, temos:

Teorema 2 *Sejam $M, N_1, N_2, N_3 \in \Lambda$. Se $M \to_\beta^* N_1$ e $M \to_\beta^* N_2$, então existe um termo N_3 tal que $N_1 \to_\beta N_3$ e $N_2 \to_\beta N_3$. Graficamente temos:*

$$\begin{tikzcd}
 & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\
N_1 \arrow[dr, "*"'] & & N_2 \arrow[dl, "*"] \\
 & N_3 &
\end{tikzcd}$$

Adicionalmente, o cálculo λ possui uma outra regra chamada η-conversão. Esta regra expressa a noção de *extensionalidade funcional*: funções que computam o mesmo valor para argumentos iguais são extensionalmente iguais. Em outras palavras, se $f(x) = g(x)$, para todo x, então $f = g$. Formalmente a regra η é dada por:

$$(\eta) \quad \lambda_x.(M\ x) \mapsto_\eta M, \text{ sempre que } x \notin FV(M) \tag{4}$$

Definimos $\to_\eta$ como sendo o fecho contextual de $\mapsto_\eta$, e $\twoheadrightarrow_\eta$ o fecho transitivo-reflexivo de $\to_\eta$. Agora se tentarmos traduzir esta afirmação para a linguagem do cálculo λ, obteremos exatamente a regra η. De fato, de acordo com nossa hipótese precisamos tomar dois λ-termos f e g que sejam "iguais" sempre que aplicados ao mesmo argumento, isto é, $f\ x =_\lambda g\ x$. Até o presente momento a única "igualdade" que temos é a obtida pela regra β (equação 3). Sendo assim, podemos escrever

$$(\lambda_a.M\ a)\ x \to_\beta (M\ a)\{a/x\} = M\ x$$

onde o segundo passo acima se justifica pelo fato de que $x \notin FV(M)$.

1.1 O Poder de Expressividade do cálculo λ

O cálculo λ é expressivo o suficiente para representar qualquer função computável. Isto é, o cálculo λ é tão expressivo quanto as máquinas de Turing. Nesta seção apresentaremos alguns exemplos de computação com o cálculo λ. Para maiores detalhes, veja por exemplo [1] Por exemplo, operações aritméticas básicas podem ser realizadas utilizando os chamados *numerais de Church*. Os numerais de Church são λ-termos utilizados para codificar os números naturais, de forma que $C_0, C_1, \ldots, C_n$ representam respectivamente 0,1, ..., n. Os numerais de Church são dados por:

$$C_0 = \lambda_{fx}.x$$
$$C_1 = \lambda_{fx}.f\ x$$
$$\ldots$$
$$C_n = \lambda_{fx}.f^n\ x$$

onde $f^n\ x = \begin{cases} x, & \text{se } n = 0; \\ f(f^{n-1}\ x), & n > 0. \end{cases}$

Lema 2

1. *$(C_n\ x)^m\ y = x^{n*m}\ y$;*
2. *$(C_n)^m\ x = C_{n^m}\ x$, para $m > 0$.*

Proposição 1 (J. B. Rosser) *Defina*

$$A_+ = \lambda_{xypq}.xp(ypq)$$

$$A_* = \lambda_{xyz}.x(yz)$$

$$A_{exp} = \lambda_{xy}.yx$$

Então valem as seguintes igualdades:

1. *$A_+\ C_n\ C_m = C_{n+m}$*
2. *$A_*\ C_n\ C_m = C_{n*m}$*
3. *$A_{exp}\ C_n\ C_m = C_{n^m}$, exceto para $m = 0$.*

1.2 Exercícios

1. Mostre que:
   (a) $A_+\ C_{33}\ C_{12} = C_{45}$
   (b) $A_*\ C_3\ C_4 = C_{12}$
   (c) $A_{exp}\ C_2\ C_3 = C_8$

2. Escreva em detalhes a prova do Lema 1.

3. Mostre que se $M, N, L \in \Gamma$, e $x \neq y$, $x \notin FV(L)$ e $y \notin FV(N)$ então:

$$M\{N/x\}\{L/y\} = M\{L/y\}\{N/x\}$$

4. Os booleanos `true` e `false` são definidos no cálculo λ respectivamente por $\lambda_{xy}.x$ e $\lambda_{xy}.y$ Defina a função `iszero` que testa se um numeral de Church é zero ou não, i.e., `iszero` $C_0 \twoheadrightarrow_\beta$ `true` e `iszero` $C_n \twoheadrightarrow_\beta$ `false` para todo $n > 0$. Note que o condicional `if` M `then` N `else` L pode ser representado simplesmente por $M\ N\ L$. De fato, `true` $N\ L = (\lambda_{xy}.x)N\ L \twoheadrightarrow_\beta N$ e `true` $N\ L = (\lambda_x y.y)N\ L \twoheadrightarrow_\beta L$. Agora defina a função predecessor:
   $\texttt{pred}\ C_k = \begin{cases} C_0, & \text{se } k = 0 \\ C_{k-1}, & \text{caso contrário} \end{cases}$
   Funções recursivas podem ser definidas no cálculo λ com a ajuda de um operador de ponto fixo. Por exemplo, o λ-termo
   $$Y = \lambda_x.(\lambda_y.y(x\ x))(\lambda_y.y(x\ x))$$
   é chamado de operador de ponto fixo porque $Y\ t \twoheadrightarrow_\beta Y(Y\ t)$ para qualquer λ-termo t. Utilizando estas informações defina a função fatorial no cálculo λ.

2 O Cálculo λ Simplesmente Tipado

Agora que conhecemos um pouco mais do cálculo λ podemos retormar o estudo do isomorfismo de Curry-Howard a partir do que foi visto antes da Seção 1. A ideia é estabelecer de maneira formal uma relação existente entre Lógica (Fragmento Implicacional da Lógica Proposicional Intuicionista) e Computação (Cálculo λ).

Definimos um *contexto* como sendo um conjunto finito de declaração de tipos para variáveis. A princípio trabalharemos com um linguagem sem constantes, onde os tipos são construídos a partir de um conjunto enumerável $\mathbb{V}$ de variáveis ou a implicação entre dois tipos já construídos:

$$\textbf{Tipos Simples} \quad \sigma ::= \mathbb{V} \mid (\sigma \to \sigma)$$

Os termos são construídos com na Seção 1, mas as variáveis são tipadas explicitamente:

$$\textbf{Termos} \quad t ::= x \mid (t\ t) \mid (\lambda_{x:\sigma}.t)$$

Temos três regras de derivação. A primeira, a regra `(var)`, permite derivar uma declaração de tipo que esteja no contexto. A regra `(app)` permite derivar o tipo do resultado de se aplicar uma função a um argumento, e a regra `(abs)` nos permite derivar o tipo de uma função:

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}\ \texttt{(var)}$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash (t\ u) : \tau}\ \texttt{(app)}$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\lambda_{x:\sigma}.t) : \sigma \to \tau}\ \texttt{(abs)}$$

A analogia com as regras (Ax), $(\to_e)$ e $(\to_i)$ vistas anteriormente é imediata:

Notação 1 *Se Γ denota um contexto então $|\Gamma|$ representa o conjunto obtido de Γ após apagar todas as informações sobre as variáveis de termos.*

Desta forma, se $\Gamma = \{x_1 : \sigma, \ldots, x_n : \sigma_n\}$ então $|\Gamma| = \{\sigma_1, \ldots, \sigma_n\}$.

$$\frac{}{\Gamma \cup \{x : \tau\} \vdash x : \tau}\ \texttt{(var)} \qquad \frac{}{|\Gamma| \cup \{\tau\} \vdash \tau}\ (\mathrm{Ax})$$

$$\frac{\Gamma \vdash t : \sigma \to \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash (t\ u) : \tau}\ \texttt{(app)} \qquad \frac{|\Gamma| \vdash \sigma \to \tau \qquad |\Gamma| \vdash \sigma}{|\Gamma| \vdash \tau}\ (\to_e)$$

$$\frac{\Gamma \cup \{x : \sigma\} \vdash t : \tau}{\Gamma \vdash (\lambda_{x:\sigma}.t) : \sigma \to \tau}\ \texttt{(abs)} \qquad \frac{|\Gamma| \cup \{\sigma\} \vdash \tau}{|\Gamma| \vdash \sigma \to \tau}\ (\to_i)$$

O isomorfismo de Curry-Howard é dado pelo seguinte teorema:

Teorema 3 (Curry-Howard)

1. *Se $\Gamma \vdash t : \tau$ então $|\Gamma| \vdash \tau$.*
2. *Se $\{\sigma_1, \ldots, \sigma_n\} \vdash \tau$ então existe um λ-termo t e variáveis distintas $x_1, x_2, \ldots, x_n$ tais que $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash t : \tau$.*

Prova. Exercício. □

Podemos estender o fragmento implicacional para uma gramática que inclua também a conjunção e disjunção. Uma prova da conjunção $\sigma_1 \wedge \sigma_2$, por exemplo, é dada por um par $\langle t_1, t_2 \rangle$, onde t_i corresponde a uma prova de σ_i $(i = 1, 2)$. Formalmente temos as seguintes correspondências:

$$\frac{\Gamma \vdash t_1 : \sigma_1 \qquad \Gamma \vdash t_2 : \sigma_2}{\Gamma \vdash \langle t_1, t_2 \rangle : \sigma_1 \wedge \sigma_2}\ \texttt{(pair)} \qquad \frac{|\Gamma| \vdash \sigma_1 \qquad |\Gamma| \vdash \sigma_2}{|\Gamma| \vdash \sigma_1 \wedge \sigma_2}\ (\wedge_i)$$

$$\frac{\Gamma \vdash t : \sigma_1 \wedge \sigma_2}{\Gamma \vdash \pi_1 t : \sigma_1}\ \texttt{(proj1)} \qquad \frac{|\Gamma| \vdash \sigma_1 \wedge \sigma_2}{|\Gamma| \vdash \sigma_1}\ (\wedge_e)$$

$$\frac{\Gamma \vdash t : \sigma_1 \wedge \sigma_2}{\Gamma \vdash \pi_2 t : \sigma_2}\ \texttt{(proj2)} \qquad \frac{|\Gamma| \vdash \sigma_1 \wedge \sigma_2}{|\Gamma| \vdash \sigma_2}\ (\wedge_e)$$

$$\frac{\Gamma \vdash t : \sigma_1}{\Gamma \vdash \texttt{left}\ t : \sigma_1 \vee \sigma_2}\ \texttt{(left)} \qquad \frac{|\Gamma| \vdash \sigma_1}{|\Gamma| \vdash \sigma_1 \vee \sigma_2}\ (\vee_i)$$

$$\frac{\Gamma \vdash t : \sigma_2}{\Gamma \vdash \texttt{right}\ t : \sigma_1 \vee \sigma_2}\ \texttt{(right)} \qquad \frac{|\Gamma| \vdash \sigma_2}{|\Gamma| \vdash \sigma_1 \vee \sigma_2}\ (\vee_i)$$

$$\frac{\Gamma \vdash t : \sigma_1 \vee \sigma_2 \qquad \Gamma \cup \{x : \sigma_1\} \vdash u : \tau \qquad \Gamma \cup \{y : \sigma_2\} \vdash v : \tau}{\Gamma \vdash (\texttt{match}\ t\ \texttt{with}\ (\texttt{left}\ x) \to u \mid (\texttt{right}\ y) \to v) : \tau}\ \texttt{(case)}$$

$$\frac{|\Gamma| \vdash \sigma_1 \vee \sigma_2 \qquad |\Gamma| \cup \{\sigma_1\} \vdash \tau \qquad |\Gamma| \cup \{\sigma_2\} \vdash \tau}{|\Gamma| \vdash \tau}\ (\vee_e)$$

Exercícios

1. Construa (em papel e lápis) as derivações completas dos exercícios do arquivo `exercicio1.v`
2. Construa (em papel e lápis) as derivações completas dos exercícios do arquivo `exercicio2.v`

3 O Cálculo λ de segunda ordem - Polimorfismo

Neste capítulo estudaremos uma forma de estender o isomorfismo de Curry-Howard para uma lógica que contenha quantificação universal sobre as variáveis de tipo.

No sistema de tipos simples visto anteriormente a abstração é feita apenas no nível dos termos: a partir de um termo t podemos "abstrair" as ocorrências da variável x de tipo σ, e construir a função de parâmetro x e corpo t denotada por $\lambda_{x:\sigma}.t$. Desta maneira, podemos construir a função identidade sobre os números naturais: $\lambda_{x:\texttt{nat}}.x : \texttt{nat} \to \texttt{nat}$; ou a função identidade sobre os booleanos: $\lambda_{x:\texttt{bool}}.x : \texttt{bool} \to \texttt{bool}$. Desta forma, precisamos construir uma função identidade para cada tipo. O mais conveniente seria construirmos uma função identidade genérica com tipo da forma $\forall \alpha.\alpha \to \alpha$ de forma que ao instanciarmos a variável universal α com `nat`(resp. `bool`) obtemos a função identidade em `nat`(resp. `bool`).

Iniciaremos introduzindo um novo construtor para os tipos:

$$\tau ::= \mathbb{V} \mid \ (\tau \to \tau) \mid (\forall_{\mathbb{V}}.\tau) \tag{5}$$

Pensando no isomorfismo de Curry-Howard, como seria a estrutura de um termo com tipo $\forall_{\mathbb{V}}.\tau$? Uma alternativa é definir uma nova abstração sobre variáveis de tipos na construção de termos:

$$\frac{\alpha \vdash \lambda_{x:\alpha}.x : \alpha \to \alpha}{\vdash \lambda_{\alpha}.\lambda_{x:\alpha}.x : \forall_{\alpha}.\alpha \to \alpha}$$

Esta construção parece promissora, mas precisamos associar um tipo à variável de tipo α. Denotaremos por $\star$ o "tipo de todos os tipos", e a derivação acima poderia ser reescrita da seguinte forma:

$$\frac{\alpha : \star \vdash \lambda_{x:\alpha}.x : \alpha \to \alpha}{\vdash \lambda_{\alpha:\star}.\lambda_{x:\alpha}.x : \forall_{\alpha}.\alpha \to \alpha}$$

Nesta construção, a hipótese $\alpha : \star$ diz que α é um tipo válido. Adicionalmente, observe que a função polimórfica $\lambda_{\alpha:\star}.\lambda_{x:\alpha}.x$ que construímos depende da variável de tipos α. Por esta razão, dizemos que os termos que construiremos dependem dos tipos[2]. Para construirmos um tipo válido precisamos declarar todas as variáveis que ocorrem no tipo a ser construído. Esta ideia sugere que contextos aqui sejam listas de declaração de tipos para variáveis, ao invés de conjuntos como no caso dos tipos simples porque a ordem das declaração agora é relevante. Adicionalmente, note que temos dois tipos de variáveis: variáveis de termos (cujo conjunto será denotado por V), e variáveis de tipos (cujo conjunto será denotado por $\mathbb{V}$). Se $\Gamma = \{x_1 : \alpha_1, x_2 : \alpha_2, \ldots, x_n : \alpha_n\}$, então $Dom(\Gamma) = \{x_1, x_2, \ldots, x_n\}$.

Definição 5 ($\lambda 2$-contexto) *Um contexto no cálculo λ de segunda ordem é definido recursivamente como a seguir:*

1. *$\emptyset$ é um $\lambda 2$-contexto, o contexto vazio;*
2. *Se Γ é um $\lambda 2$-contexto, $\alpha \in \mathbb{V}$ e $\alpha \notin Dom(\Gamma)$ então $\Gamma, \alpha : \star$ é um $\lambda 2$-contexto.*
3. *Se Γ é um $\lambda 2$-contexto, $\alpha \in \mathbb{T}_2$ tal que todas as variáveis livres θ que ocorrem em α estão em $Dom(\Gamma)$, $x \in V$ e $x \notin Dom(\Gamma)$ então $\Gamma, x : \alpha$ é um $\lambda 2$-contexto.*

Obtemos assim, a seguinte gramática para tipos e termos, respectivamente:

$$\begin{aligned} \mathbb{T}_2 &::= \mathbb{V} \mid \ (\mathbb{T}_2 \to \mathbb{T}_2) \mid (\forall_{\mathbb{V}:\star}.\mathbb{T}_2) \\ \Lambda_2 &::= V \mid \ (\Lambda_2\ \Lambda_2) \mid (\Lambda_2\ \mathbb{T}_2) \mid (\lambda_{V:\mathbb{T}_2}.\Lambda_2) \mid (\lambda_{\mathbb{V}:\star}.\Lambda_2) \end{aligned} \tag{6}$$

[2] *Terms depending on types*

O sistema obtido é conhecido como *cálculo λ de segunda ordem* cujas regras de derivação são dadas a seguir:

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x:\tau}\ \texttt{(var)}$$

$$\frac{\Gamma \cup \{x:\sigma\} \vdash t:\tau}{\Gamma \vdash (\lambda_{x:\sigma}.t):\sigma \to \tau}\ \texttt{(abs)}$$

$$\frac{\Gamma \vdash t:\sigma \to \tau \qquad \Gamma \vdash u:\sigma}{\Gamma \vdash (t\ u):\tau}\ \texttt{(app)}$$

$$\frac{}{\Gamma \vdash B:\star}\ \texttt{(form)},\ \text{if } (\Box)$$

$$\frac{\Gamma \cup \{\sigma:\star\} \vdash t:\tau}{\Gamma \vdash (\lambda_{\sigma:\star}.t):\forall_{\sigma:\star}.\tau}\ \texttt{(abs2)}$$

$$\frac{\Gamma \vdash t:\forall_{\sigma:\star}.\tau \qquad \Gamma \vdash \theta:\star}{\Gamma \vdash (t\ \theta):\tau[\sigma := \theta]}\ \texttt{(app2)}$$

onde $(\Box)$ significa que Γ é um $\lambda 2$-contexto, $B \in \mathbb{T}_2$ e todas as variáveis livres de B estão declaradas em Γ.

Podemos construir a função identidade nos naturais da seguinte forma:

$$\frac{\dfrac{\alpha:\star, a:\alpha \vdash a:\alpha}{\alpha:\star \vdash \lambda_{a:\alpha}.a:\alpha \to \alpha}\ \texttt{(abs)}}{\vdash \lambda_{\alpha:\star}\lambda_{a:\alpha}.a:\forall\alpha.\alpha \to \alpha}\ \texttt{(abs2)}$$

$$\frac{\dfrac{\dfrac{nat:\star, \alpha:\star, a:\alpha \vdash a:\alpha}{nat:\star, \alpha:\star \vdash \lambda_{a:\alpha}.a:\alpha \to \alpha}\ \texttt{(abs)}}{nat:\star \vdash \lambda_{\alpha:\star}\lambda_{a:\alpha}.a:\forall\alpha.\alpha \to \alpha}\ \texttt{(abs2)} \qquad nat:\star \vdash nat:\star}{nat:\star \vdash \lambda_{a:nat}.a:nat \to nat}\ \texttt{(app2)}$$

Exercícios

1. Quantos $\lambda 2$-contextos podemos construir a partir das declarações $\alpha:\star$, $\beta:\star$, $f:\alpha \to \beta$ e $x:\alpha$?
2. Um termo t é dito *legal* se existem um $\lambda 2$-contexto e $\sigma \in \mathbb{T}_2$ tal que $\Gamma \vdash t:\sigma$. Mostre que o termo $\lambda_{\alpha,\beta,\gamma:\star}.\lambda_{f:\alpha\to\beta}.\lambda_{g:\beta\to\gamma}.\lambda_{x:\alpha}.(g\ (f\ x))$ é legal.
3. Sejam $\perp \equiv \forall_{\alpha:\star.\alpha}$ e $\Gamma = \beta:\star, x:\perp$.
   (a) Mostre que $\perp$ é legal.
   (b) Construa um termo de tipo β no contexto Γ. Ou seja, encontre um habitante do tipo β.

(c) Encontre três habitantes distintos e em forma normal[3] do tipo $\beta \to \beta$ no contexto Γ.

(d) Mostre que os termos $\lambda_{f:\beta\to\beta\to\beta}.f(x\ \beta)(x\ \beta)$ e $x\ ((\beta \to \beta \to \beta) \to \beta)$ têm o mesmo tipo no contexto Γ.

4. Uma versão dos numerais de Church para o cálculo $\lambda 2$ pode ser dada como a seguir:

   (a) $\mathtt{nat} \equiv \forall_{\alpha:\star}.(\alpha \to \alpha) \to \alpha \to \alpha$

   (b) $Zero \equiv \lambda_{\alpha:\star}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.x$ de tipo $\mathtt{nat}$;

   (c) $One \equiv \lambda_{\alpha:\star}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f\ x$ de tipo $\mathtt{nat}$;

   (d) $Two \equiv \lambda_{\alpha:\star}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\alpha}.f(f\ x)$ de tipo $\mathtt{nat}$;

   Defina a função sucessor como $Succ \equiv \lambda_{n:\mathtt{nat}}.\lambda_{\beta:\star}.\lambda_{f:\beta\to\beta}.\lambda_{x:\beta}.f(n\ \beta\ f\ x)$. Mostre que $Succ$ funciona como esperado mostrando que $Succ\ Zero =_\beta One$ e $Succ\ One =_\beta Two$.

   Defina $Add \equiv \lambda_{mn:\mathtt{nat}}.\lambda_{\alpha:\star}.\lambda_{f:\alpha\to\alpha}.\lambda_{x:\mathtt{nat}}.m\ \alpha\ f(n\ \alpha\ f\ x)$, e mostre que $Add\ One\ One =_\beta Two$.

   Defina um $\lambda 2$ que simule a operação de multiplicação.

Referências

[1] M. Ayala-Rincón and F. L. C. de Moura. *Fundamentos da Programação Lógica e Funcional - O princípio de Resolução e a Teoria de Reescrita*. Universidade de Brasília, 2014.

[2] H. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Simbolic Logic*, 3(3):181–215, 1997.

[3] A. Church. A set of postulates for the foundation of logic. *Annals of Math.*, 33(2):346–366, 1932.

[4] A. Church. A set of postulates for the foundation of logic (second paper). *Annals of Math.*, 34(2):839–864, 1933.

[5] A. Church and J. B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39:472–482, 1936.

[6] S. Kleene and B. Rosser. The inconsistency of certain formal logics. *Annals of Math.*, 36(2):630–636, 1935.

[7] B. A. W. Russel and A. N. Whitehead. *Principia Mathematica*, volume 1 and 2. Cambridge University Press, 1910-1913.

[3] Um termo está em forma normal se não puder ser reduzido pela regra β.