Tipos, Categorias e Lógicas

Notas em Teoria dos Tipos, Teoria das Categorias e Lógica.

edição 0.0
Autores: Mateus Galdino
2025-01-15

Contents

1 Prefácio

A Teoria dos Tipos é uma área nova crescente de ampla interseção com outras áreas também novas ou áreas já consolidadas. Essa interseção dá frutos práticos interessantes para as áreas envolvidas. Por exemplo: no campo da computação, a maioria das linguagens de programação possui tipagem e, sem entrar no mérito das diferentes abordagens de tipagem para cada linguagem, é possível descrever a tipagem delas utilizando uma teoria dos tipos adequada. Já no campo da matemática, é possível perceber que as teorias dos tipos descritas aqui possuem modelos conhecidos na teoria das categorias que permitem formulações de objetos matemáticos já conhecidos (como Grupos, Espaços Topológicos, etc). No meio desses dois exemplos, existe a tentativa de aproximar a computação dos fundamentos da matemática a partir de assistentes de prova.

Essas notas foram escritas por dois fins. O primeiro é expor em lingua portuguesa a vasta gama de conceitos explorados na teoria dos tipos, deixando essa área da matemática e da computação o mais acessível possível para iniciantes vindos de diversos ambientes. Em lingua inglesa, já existem várias fontes possíveis para adentrar essa teoria, que serão referenciadas a partir dessas notas, mas em português as poucas fontes que existem estão em dissertações acadêmicas pouco preocupadas com a difusão das ideias para fora de seus nichos. O segundo fim é, em certa medida, conseguir, através dessa exposição, que mais e mais pessoas tenham interesse pelo assunto e começem a pesquisar, visto que nos centros e depertamentos brasileiros, sejan de matemática ou de computação, essa área recebe pouca a nenhuma atenção, já que os professores especializados nesses assuntos já não estão comprometidos a ensinar os alunos de graduação essa área. Dessa forma, essas notas também se colocam como um desafio: ensinar o máximo de teoria dos tipos possível para alunos de graduação, supondo o mínimo matematicamente.

Essas notas então podem ser utilizadas sem dúvida por professores que queiram se aventurar no ensino da teoria dos tipos.

A primeira parte tenta desenvolver as diversas teorias de tipos denominadas de λ-cubo. Essa primeira parte usa como base (o primeiro subcapítulo de cada capítulo) o livro *Type Theory and Formal Proof* de Nederpelt e Geuvers, mas adentra tópicos mais profundos em cada teoria a partir dos outros subcapítulos.

Já a segunda parte desenvolve outras construções paralelas ao λ-cubo, derivadas do λ-calculo não-tipado, como o λμ-cálculo e o κ-cálculo. Cada cálculo é retirado de artigos diferentes e compilados no mesmo lugar.

A parte três desenvolve a teoria das categorias necessária para a semântica de cada uma das teorias dos tipos desenvolvidas nas partes I e II, desenvolvendo o conceito de categorias até a teoria dos Topos e construções paralelas. Essa parte é bastante influenciada pelo livro *Introduction to Higher Order Categorical Logic* de Lambek e Scott e pelo livro *Sheaf Theory Through Examples* de Daniel Rosiak.

A parte IV entra nas diversas teorias homotópicas de tipos, desde sua precursora, a *Teoria dos Tipos de Martin-Löf*, e a original do livro *Homotopy Type Theory* até construções mais recentes. A maioria dessas teorias está espalhada em diversos artigos, então o trabalho aqui se torna compilá-las em um único lugar de forma a criar um fio condutor entre elas.

A parte V dessenvolve a semântica categorial das HoTT utilizando conceitos

da teoria das ∞-categorias, teoria das homotopias (em suas versões simpliciais e cúbicas) e conceitos já trabalhados na parte III.

A parte VI é a parte final e serve como exposição de definições voltadas para a lógica e a teoria da prova, com a exposição do cálculo de sequêntes, da dedução natural e de outras áreas correlatas. Essa parte trás inspiração no livro *Logic and Structure* do Dirk van Dalen e *An Introduction to Proof Theory* de Galvan et al.

Links importantes:

- Caso o leitor encontre algum erro ou problema nas notas, por favor avisar em <https://github.com/MateusGaldinoLG/notasTT/issues>.
- Caso o leitor queira contribuir no geral com adição ou escrita de temas: <https://github.com/MateusGaldinoLG/notasTT>
- Para verificar o progresso da escrita do livro: <https://github.com/MateusGaldinoLG/notasTT/blob/main/passos.md>

Part I
O Cubo Lambda

2 Cálculo Lambda não-tipado ($\lambda_{\beta\eta}$)

A teoria dos tipos possui como história de origem algumas tentativas falhas. O conceito de tipos pode ser mapeado para dois matemáticos importantes que fizeram usos bem diferentes dele: Bertrand Russel (e Walfred North Whitehead) na Principia Mathematica e Alonzo Church no seu Cálculo λ simplesmente tipado (STλC).

A teoria dos tipos que é usada hoje, provém do segundo autor e de outros autores que vêm dessa tradição. Por isso, o início dessas notas se propõe a começar do básico, definindo o que é o Cálculo λ não tipado e quais questões levaram Church a desenvolver a teoria dos tipos em cima dele.

Aqui, será traduzido "λ-calculus" como "Cálculo λ", decisão que perde a estética do hífen, mas que mantém a unidade com outras traduções de "X calculus" no corpo matemático brasileiro, como o "Cálculo Diferencial e Integral", o "Cálculo de sequentes", o "Cálculo de variações", etc.

2.1 O Cálculo

2.1.1 Definições

O cálculo lambda serve como uma abstração em cima do conceito de função. Uma função é uma estrutura que pega um *input* e retorna um *output*, por exemplo a função $f(x) = x^2$ pega um input x e retorna seu valor ao quadrado x^2. No cálculo lambda, essa função pode ser denotada por $\lambda x.x^2$, onde $\lambda x.$ simboliza que essa função espera receber como entrada x. Quando se quer saber qual valor a função retorna para uma entrada específica, são usados números no lugar das variáveis, como por exemplo $f(3) = 3^2 = 9$. No cálculo lambda, isso é feito na forma de $(\lambda x.x^2)(3)$.

Esses dois principrios de construção são definidos como:

- **Abstração**: Seja M uma expressão e x uma variável, podemos construir uma nova expressão $\lambda x.M$. Essa expressão é chamada de Abstração de x sobre M
- **Aplicação**: Sejam M e N duas es expressões, podemos construir uma expressão MN. Essa expressão é chamada de Aplicação de M em N.

Dadas essas operações, é preciso também de uma definição que dê conta do processo de encontrar o resultado após a aplicação em uma função. Esse processo é chamado de β-redução. Ela faz uso da substituição e usa como notação os colchetes.

Definição 2.1 (β-redução)**.** A β-redução é o processo de resscrita de uma expressão da forma $(\lambda x.M)N$ em outra expressão $M[x := N]$, ou seja, a expressão M na qual todo x foi substituido por N.

2.1.2 Sintáxe do Cálculo Lambda

É interessante definir a sintaxe do cálculo lambda de forma mais formal. Para isso, são utilizados métodos que podem ser familiares para aqueles que já trabalharam com lógica proposicional, lógica de primeira ordem ou teoria de modelos.

Primeiro, precisamos definir a linguagem do Cálculo λ.

Definição 2.2. (i) Os *termos lambda* são palavras em cima do seguinte alfabeto:

- variáveis: $v_0, v_1, \ldots$
- abstrator: λ
- parentesis: $(,)$

(ii) O conjunto de λ-termos Λ é definido de forma indutiva da seguinte forma:

- Se x é uma variável, então $x \in \Lambda$
- $M \in \Lambda \rightarrow (\lambda x.M) \in \Lambda$
- $M, N \in \Lambda \rightarrow MN \in \Lambda$

Na teoria dos tipos e no cálculo lambda, é utilizada uma forma concisa de definir esses termos chamada de Formalismo de Backus-Naur ou Forma Normal de Backus (BNF, em inglês). Nessa forma, a definição anterior é reduzida à:

$$\Lambda = V|(\Lambda\Lambda)|(\lambda V\Lambda)$$

Onde V é o conjunto de variáveis $V = \{x, y, z, \ldots\}$

Para expressar igualdade entre dois termos de Λ utilizamos o simbolo $\equiv$.

Algumas definições indutivas podem ser formadas a partir da definição dos λ-termos.

Definição 2.3 (Multiconjunto de subtermos)**.**

1. (Base) $\mathtt{Sub}(x) = \{x\}$, para todo $x \in V$
2. (Aplicação) $\mathtt{Sub}((MN)) = \mathtt{Sub}(M) \cup \mathtt{Sub}(N) \cup \{(MN)\}$
3. (Abstração) $\mathtt{Sub}((\lambda x.M)) = \mathtt{Sub}(M) \cup \{(\lambda x.M)\}$

Observações:
(i) Um subtermo pode ocorrer múltiplas vezes, por isso é escolhido chamar de multiconjunto (ii) A abstração de vários termos ao mesmo tempo pode ser escrita como $\lambda x.(\lambda y.x)$ ou como $\lambda xy.x$.

Lema 2.1 (Propriedades de $\mathtt{Sub}$)**.**

- (Reflexividade) Para todo λ-termo M, temos que $M \in \mathtt{Sub}(M)$
- (Transitividade)Se $L \in \mathtt{Sub}(M)$ e $M \in \mathtt{Sub}(N)$, então $L \in \mathtt{Sub}(N)$.

Definição 2.4 (Subtermo próprio)**.** L é um subtermo próprio de M se L é subtermo de M e $L \not\equiv M$

Exemplos:

1. Seja o termo $\lambda x.\lambda y.xy$, vamos calcular seus subtermos:

$$\begin{aligned}\mathtt{Sub}(\lambda x.\lambda y.xy) &= \{\lambda x.\lambda y.xy\} \cup \mathtt{Sub}(\lambda y.xy) \\ &= \{\lambda x.\lambda y.xy\} \cup \{\lambda y.xy\} \cup \mathtt{Sub}(xy) \\ &= \{\lambda x.\lambda y.xy\} \cup \{\lambda y.xy\} \cup \mathtt{Sub}(x) \cup \mathtt{Sub}(y) \\ &= \{\lambda x.\lambda y.xy, \lambda y.xy, x, y\}\end{aligned}$$

2. Seja o termo $(y(\lambda x.(xyz)))$, vamos calcular os seus subtermos:

$$\begin{aligned}\mathtt{Sub}(y(\lambda x.(xyz))) &= \mathtt{Sub}(y) \cup \mathtt{Sub}((\lambda x.(xyz))) \\ &= \{y\} \cup \{(\lambda x.(xyz))\} \cup \mathtt{Sub}((xyz)) \\ &= \{y\} \cup \{(\lambda x.(xyz))\} \cup \mathtt{Sub}(x) \cup \mathtt{Sub}(y) \cup \mathtt{Sub}(z) \\ &= \{y\} \cup \{(\lambda x.(xyz))\} \cup \{x\} \cup \{y\} \cup \{z\} = \{y, (\lambda x.(xyz)), x, y, z\}\end{aligned}$$

Outro conjunto importante para a sintaxe do cálculo lambda é o de variáveis livres. Uma variável é dita *ligante* se está do lado do λ. Em um termo $\lambda x.M$, x é uma variável ligante e toda aparição de x em M é chamada de *ligada*. Se existir uma variável em M que não é ligante, então dizemos que ela é *livre*. Por exemplo, em $\lambda x.xy$, o primeiro x é ligante, o segundo x é ligado e y é livre.

O conjunto de todas as variáveis livres em um termo é denotado por FV e definido da seguinte forma:

Definição 2.5 (Multiconjunto de variáveis livres)**.**

1. (Base) $FV(x) = \{x\}$, para todo $x \in V$
2. (Aplicação) $FV((MN)) = FV(M) \cup FV(N) \cup \{(MN)\}$
3. (Abstração) $FV((\lambda x.M)) = FV(M) \setminus \{x\}$

Exemplos:

1. Seja o termo $\lambda x.\lambda y.xyz$, vamos calcular seus subtermos:

$$\begin{aligned}FV(\lambda x.\lambda y.xyz) &= FV(\lambda y.xyz) \setminus \{x\} \\ &= FV(xyz) \setminus \{y\} \setminus \{x\} \\ &= FV(x) \cup FV(y) \cup FV(z) \setminus \{y\} \setminus \{x\} \\ &= \{x, y, z\} \setminus \{y\} \setminus \{x\} \\ &= \{z\}\end{aligned}$$

Vamos definir os termos fechados da seguinte forma:

Definição 2.6. O λ-termo M é dito *fechado* se $FV(M) = \emptyset$. Um λ-termo fechado também é chamado de *combinador*. O conjunto de todos os λ-termos fechados é chamado de Λ^0.

Os combinadores são muito utilizados na *Lógica Combinatória*, mas vamos explorá-los mais a frente.

2.1.3 Conversão

No cálculo Lambda, é possível renomear variáveis ligantes/ligadas, pois a mudança dos nomes dessas variáveis não muda a sua interpretação. Por exemplo, $\lambda x.x^2$ e $\lambda u.u^2$ podem ser utilizadas de forma igual, mesmo que com nomes diferentes. A Renomeação será definida da seguinte forma:

Definição 2.7. Seja $M^{x\to y}$ o resultado da troca de todas as livre-ocorrencias de x em M por y. A relação de renomeação é expressa pelo símbolo $=_\alpha$ e é definida como: $\lambda x.M =_\alpha \lambda y.M^{x\to y}$, dado que $y \notin FV(M)$ e y não seja ligante em M.

Podemos extender essa definição para a definição do renomeamento, chamado de α-conversão.

Definição 2.8 (α-conversão)**.**

1. (Renomeamento) $\lambda x.M =_\alpha \lambda y.M^{x\to y}$
2. (Compatibilidade) Sejam M, N, L termos. Se $M =_\alpha N$, então $ML =_\alpha NL$, $LM =_\alpha LN$
3. (Regra ξ) Para um z qualquer, se $M =_\alpha N$, então $\lambda z.M =_\alpha \lambda z.N$
4. (Reflexividade) $M =_\alpha M$
5. (Simetria) Se $M =_\alpha N$, então $N =_\alpha M$
6. (Transitividade) Se $L =_\alpha M$ e $M =_\alpha N$, então $L =_\alpha N$

A partir dos pontos (4), (5) e (6) dessa definição, é possível dizer que a α-conversão é uma relação de equivalência, chamada de α-equivalência.

Exemplos:

1. $(\lambda x.x(\lambda z.xy)) =_\alpha (\lambda u.u(\lambda z.uy))$
2. $(\lambda x.xy) \neq_\alpha (\lambda y.yy)$

2.1.4 Substituição

Podemos definir agora a substituição de um termo por outro da seguinte forma:

Definição 2.9 (Substituição)**.**

1. $x[x := N] \equiv N$
2. $y[y := x] \equiv y$, se $x \not\equiv y$
3. $(PQ)[x := N] \equiv (P[x := N])(Q[x := N])$
4. $(\lambda y.P)[x := N] \equiv (\lambda z.P^{y\to z})[x := N]$ se $(\lambda z.P^{y\to z})$ é α-equivalente a $(\lambda y.P)$ e $z \notin FV(N)$

A notação $[x := N]$ é uma meta-notação, pois não está definida na sintaxe do cálculo lambda. Na literatura também é possível ver a notação $[N/x]$ para definir a substituição.

2.1.5 Beta redução

Voltando à aplicação, agora com a substituição em mente, podemos dizer que a aplicação de um termo N em $\lambda x.M$, na forma de $(\lambda x.M)N$ é a mesma coisa que $M[x := N]$. Nesse caso, essa única substituição entre termos pode ser descrita na seguinte definição:

Definição 2.10 (β-redução para único passo)**.**

1. (Base) $(\lambda x.M)N \rightarrow_\beta M[x := N]$
2. (Compatibilidade) Se $M \rightarrow_\beta N$, então $ML \rightarrow_\beta NL$, $LM \rightarrow_\beta LN$ e $\lambda x.M \rightarrow_\beta \lambda x.N$

O termo $(\lambda x.M)N$ é chamado de *redex*, vindo do ingles "reducible expression" (expressão reduzível), e o subtermo $M[x := N]$ é chamado de *contractum* do redex.

Exemplos:

1. $(\lambda x.x(xy))N \rightarrow_\beta N(Ny)$
2. $(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx)$
3. $(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda y.yv)z \rightarrow_\beta zv$

Os exemplos 2 e 3 são importantes por duas razões:

- Com o exemplo 3 é possível ver que é possível concatenar várias reduções seguindas, vamos colocar uma definição mais geral a diante que lide com isso.
- Com o exemplo 2 é possível ver que existem termos que, quando beta-reduzidos, retornam eles mesmos. Isso faz com que cálculo lámbda não tipado tenha propriedades interessantes, pois muitas vezes a simplificação não termina. Ou seja, é possível haver cadeias de beta redução que não possuem termo mais simples.

Definição 2.11 (β-redução para zero ou mais passos)**.** $M \twoheadrightarrow_\beta N$ (lê-se: M beta reduz para N em vários passos) se existe um $n \geq 0$ e existem termos M_0 até M_n tais que $M_0 \equiv M$, $M_n \equiv N$ e para todo i tal que $0 \leq i < n$:

$$M_i \rightarrow_\beta M_{i+1}$$

Ou Seja:

$$M \equiv M_0 \rightarrow_\beta M_1 \rightarrow_\beta \cdots \rightarrow_\beta M_{n-1} \rightarrow_\beta M_n \equiv N$$

Lema 2.2.

1. $\twoheadrightarrow_\beta$ é uma extensão de $\rightarrow_\beta$, ou seja: se $M \rightarrow_\beta N$, então $M \twoheadrightarrow_\beta N$
2. $\twoheadrightarrow_\beta$ é reflexivo e transitivo, ou seja:
   - (reflexividade) Para todo M, $M \twoheadrightarrow_\beta M$
   - (transitividade) Para todo L, M, e N. Se $L \twoheadrightarrow_\beta M$ e $M \twoheadrightarrow_\beta N$, então $L \twoheadrightarrow_\beta N$

Prova

1. Na definição 1.11, seja $n = 1$, então $M \equiv M_0 \rightarrow_\beta M_1 \equiv N$, que é a mesma coisa que $M \rightarrow_\beta N$
2. Se $n = 0$, $M \equiv M_0 \equiv N$
3. A transitividade também segue da definição

Uma extensão dessa β-redução geral é a β- conversão, definida como:

Definição 2.12 (β-conversão)**.** $M =_\beta N$ (lê-se: M e N são β-convertíveis) se existe um $n \geq 0$ e existem termos M_0 até M_n tais que $M_0 \equiv M$, $M_n \equiv N$ e para todo i tal que $0 \leq i < n$: Ou $M_i \rightarrow_\beta M_{i+1}$ ou $M_{i+1} \rightarrow_\beta M_i$

Lema 2.3.

1. $=_\beta$ é uma extensão de $\twoheadrightarrow_\beta$ em ambas as direções, ou seja: se $M \twoheadrightarrow_\beta N$ ou $N \twoheadrightarrow_\beta M$, então $M =_\beta N$
2. $=_\beta$ é uma relação de equivalência, ou seja, possui reflexividade, simetria e transitividade
   - (reflexividade) Para todo M, $M =_\beta M$
   - (Simetria) Para todo M e N, se $M =_\beta N$, então $N =_\beta M$
   - (transitividade) Para todo L, M, e N. Se $L =_\beta M$ e $M =_\beta N$, então $L =_\beta N$

2.1.6 Forma Normal

Podemos definir a hora de parar de reduzir, para isso vamos introduzir o conceito de forma Normal

Definição 2.13 (Forma normal β ou β-normalização)**.**

1. M está na forma normal β se M não possui nenhum redex
2. M possui uma formal normal β, ou é β-normalizável, se existe um N na forma normal β tal que $M =_\beta N$. N é chamado de a *forma normal* β de M.

Lema 2.4. Se M está em sua forma normal β, então $M \twoheadrightarrow_\beta N$ implica em $M \equiv N$

Exemplos:

1. $(\lambda x.(\lambda y.yx)z)v$ tem como formal normal β zv, pois $(\lambda x.(\lambda y.yx)z)v \rightarrow_\beta zv$ (como visto nos exemplos anteriores) e zv está na forma normal β
2. Vamos definir um termo $\Omega := (\lambda x.xx)(\lambda x.xx)$, Ω não está na forma normal β, pois pode ser β-reduzido, mas não possui também forma normal β, pois ele sempre é β-reduzido para ele mesmo.
3. Seja $\Delta := (\lambda x.xxx)$, então $\Delta\Delta \rightarrow_\beta \Delta\Delta\Delta \rightarrow_\beta \Delta\Delta\Delta\Delta \rightarrow_\beta \ldots$. Logo $\Delta\Delta$ não possui forma normal.

Definição 2.14 (Caminho de Redução)**.**
Um caminho de redução finito de M é uma sequência finita de termos $N_0, N_1, \ldots, N_n$ tais que $N_0 \equiv M$ e $N_i \to_\beta N_{i+1}$, para todo $0 \leq i < n$.
Um caminho de redução infinito de M é uma sequência infinita de termos $N_0, N_1, \ldots$ tais que $N_0 \equiv M$ e $N_i \to_\beta N_{i+1}$, para todo $i \in \mathbb{N}$

Considerando esses dois tipos de caminhos de redução, vamos definir dois tipos de normalização

Definição 2.15 (Normalização Fraca e Forte)**.**

1. M é *fracamente normalizável* se existe um N na forma normal β tal que $M \twoheadrightarrow_\beta N$

2. M é *fortemente normalizável* se não existem caminhos de requção infinitos começando de M.

Todo termo M que é fortemente normalizável é fracamente normalizável.
Os termos Ω e Δ não são nem fortemente normalizáveis, nem fracamente normalizáveis.
É possível relacionar a normalização fraca com a forma normal β usando a intuição que, se M reduz para ambos N_1 e N_2, então existe um termo N_3 que exista no caminho de redução de ambos N_1 e N_2.

Teorema 2.1 (Teorema de Church-Rosser ou Teorema da Confluência)**.**
Suponha que para um λ-termo M, tanto $M \twoheadrightarrow_\beta N_1$ e $M \twoheadrightarrow_\beta N_2$. Então existe um λ-termo N_3 tal que $N_1 \twoheadrightarrow_\beta N_3$ e $N_2 \twoheadrightarrow_\beta N_3$

A prova desse teorema pode ser encontrada no Livro de Barendregt.
Uma consequência importante desse teorema é que o resultado do calculo feito em cima do termo não depende da ordem que esses cálculos são feitos. A escolha dos redexes não interfere no resultado final.

Corolário 2.1.
Suponha que $M =_\beta N$. Então existe um termo L tal que $M \twoheadrightarrow_\beta L$ e $N \twoheadrightarrow_\beta L$.

prova. Como $M =_\beta N$, então, pela definição, existe um $n \in \mathbb{N}$ tal que:

$$M \equiv M_0 \rightleftarrows_\beta M_1 \ldots M_{n-1} \rightleftarrows_\beta M_n \equiv N$$

. Onde $M_i \rightleftarrows_\beta M_{i+1}$ significa que ou $M_i \to_\beta M_{i+1}$ ou $M_{i+1} \to_\beta M_i$. Vamos provar por indução em n:

1. Quando $n = 0$: $M \equiv N$. Então sendo $L \equiv M$, $M \twoheadrightarrow_\beta L$ e $N \twoheadrightarrow_\beta L$ (por zero passsos)

2. Quando $n = k > 0$, entao existe M_{k-1}. Logo temos que $M \equiv M_0 \rightleftarrows_\beta M_1 \ldots M_{k-1} \rightleftarrows_\beta M_k \equiv N$. Por indução, existe um L' tal que $M_0 \twoheadrightarrow_\beta L'$ e $M_{k-1} \twoheadrightarrow_\beta L'$. Vamos dividir $M_{k-1} \rightleftarrows_\beta M_k$ em dois casos

   (a) Se $M_{k-1} \to_\beta M_k$, então como $M_{k-1} \to_\beta M_k$ e $M_{k-1} \twoheadrightarrow_\beta L'$, então, pelo Teorema de Church-Rosser, existe um L tal que $L' \twoheadrightarrow_\beta L$ e $M_k \twoheadrightarrow_\beta L$. Logo encontramos L.

   (b) Se $M_k \to_\beta M_{k-1}$, então como $M_0 \twoheadrightarrow_\beta L'$ e $M_k \twoheadrightarrow_\beta L'$, L' é o próprio L.

□.

Lema 2.5.

1. Se M possui forma normal β N, então $M \twoheadrightarrow_\beta N$.
2. Um λ-termo tem no máximo uma forma normal β

Prova

1. Seja $M =_\beta N$, com N como formal normal β. Então, pelo corolário anterior, existe um L tal que $M \twoheadrightarrow_\beta L$ e $N \twoheadrightarrow_\beta L$. Como N é a forma normal, N não é mais redutível e $N \equiv L$. Então $M \twoheadrightarrow_\beta L \equiv N$, logo $M \twoheadrightarrow_\beta N$.
2. Suponha que M possui duas formas normais β N_1 e N_2. Então por (1), $M \twoheadrightarrow_\beta N_1$ e $M \twoheadrightarrow_\beta N_2$. Pelo teorema de Church-Rosser, existe um L tal que $N_1 \twoheadrightarrow_\beta L$ e $N_2 \twoheadrightarrow_\beta L$. Mas como N_1 e N_2 estão na forma normal, $L \equiv N_1$ e $L \equiv N_2$. Então pela transitividade da equivalência, $N_1 \equiv N_2$.

□.

2.1.7 Teorema do ponto fixo

No Cálculo λ, todo λ-termo L possui um *ponto fixo*, ou seja, existe um λ-termo M tal que $LM =_\beta M$. O termo Ponto Fixo vêm da análise funcional: seja f uma função, então f possui um ponto fixo a se $f(a) = a$.

Teorema 2.2. Para todo $L \in \Lambda$, existe um $M \in \Lambda$ tal que $LM =_\beta M$.

prova: Seja L um λ-termo e defina $M := (\lambda x.L(xx))(\lambda x.L(xx))$. M é um redex, logo:

$$\begin{aligned} M &\equiv (\lambda x.L(xx))(\lambda x.L(xx)) \\ &\to_\beta L((\lambda x.L(xx))(\lambda x.L(xx))) \\ &\equiv LM \end{aligned}$$

Logo $LM =_\beta M$. □

Pela prova anterior, podemos perceber que M pode ser generalizado para todo λ-termo. Esse M será denominado de *Combinador de ponto fixo* e escrito na forma:

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

2.1.8 Eta redução

A junção da definição 0.8 com as definições de β-redução gera uma teoria que será chamada aqui de λ_β. Para essa teoria, faltam alguns detalhes que podem, ou não, ser introduzidos a depender do que se precisa.

A η-redução é a segunda redução possível dentro do cálculo λ. Através dela é possível remover uma abstração que não faz nada para o termo interior. Sua definição é:

Definição 2.16 (η-redução)**.**

1. $(\lambda x.Mx) \rightarrow_\eta M$, onde $x \notin FV(M)$.

A junção da teoria λ com a η-redução será chamada aqui de $\lambda_{\beta\eta}$.

Uma outra adição possível à teoria λ é chamada de extencionalidade e definida da seguinte forma:

Definição 2.17 (extencionalidade **Ext**)**.** Dados os termos M e N, se $Mx = Nx$ para todo λ-termo x, com $x \notin FV(MN)$, então $M = N$.

Ext introduz no cálculo λ a noção presente na teoria dos conjuntos de igualdade entre funções. Na teoria dos conjuntos, duas funções $f : A \rightarrow B$ e $g : A \rightarrow B$ são iguais se, para todo $x \in A$, $f(x) = g(x)$.

A união da teoria λ com **Ext** é chamada de $\lambda +$ **Ext**.

Teorema 2.3 (Teorema de Curry)**.** As teorias $\lambda_{\beta\eta}$ e $\lambda +$ ***Ext*** são equivalentes.

Prova: Primeiro, é necessário mostrar que η é derivável de $\lambda +$ ***Ext***. Seja a igualdade $(\lambda x.Mx)x = Mx$, por **Ext**, $\lambda x.Mx = M$.

Segundo, é necessário mostrar que dado $Mx = Nx$, é possível derivar $M = N$ em $\lambda_{\beta\eta}$. Para isso, seja $Mx = Nx$, realizando ξ-redução, tem-se que $\lambda x.Mx = \lambda x.Nx$. Fazendo η-redução dos dois lados, $M = N$. □

Existe uma outra formulação da extencionalidade dentro do cálculo λ chamado de regra ω. É necessário um equivalente à **Ext** para restrições do cálculo λ que só possuem termos fechados, para isso, é desenvolvida a regra ω:

Definição 2.18 (Regra ω)**.** Dados os termos M e N, se $MQ = NQ$ para todo termo fechado Q, então $M = N$.

Da regra ω é possível deduzir **Ext**, mas não o oposto. A prova dessa dedução não será mostrada.

Posteriormente, será feito uma discussão de teorias dos tipos que aceitam **Ext** como um axioma no estilo de $\lambda +$ ***Ext*** e outras que conseguem derivar a extencionalidade através de outras propriedades, como $\lambda_{\beta\eta}$.

2.1.9 Codificações dentro do Cálculo λ

O primeiro exemplo de transformação de funções em λ-termos, $f(x) = x^2$ para $\lambda x.x^2$, pode parecer correto, mas supõe mais que foi definido até então. Pois partindo somente da sintaxe e das transformações vistas nas seções anteriores, não foi definido coisas básicas como o que significa a exponenciação ou o número 2. Se o cálculo lambda é colocado como um possível substituto para a teoria das funções baseada na teoria dos conjuntos, então ele deve ser capaz de definir todas essas coisas de forma interna. Por isso, foram desenvolvidas as *codificações*, das quais a primeira e mais conhecida é a *Codificação de Church* (Church Encoding).

Primeiro, é necessário definir os números naturais e, para isso, é preciso de combinadores que traduzam os axiomas de Peano para os números naturais. Ou seja, precisamos definir o número 0 e a função sucessor $suc(x) = x+1$. Para isso, diferente das outras definições indutivas vistas anteriormente, primeiro serão definidos os números e depois as operações.

Definição 2.19 (Numerais de Church)**.**

1. $zero := \lambda fx.x$

2. $um := \lambda fx.fx$

3. $dois := \lambda fx.f(fx)$
   ...

4. $n := \lambda fx.f^n x$

Onde $f^n x$ é $f(f(f \ldots x))$ n vezes.
As operações são descritas na forma:

Definição 2.20 (Operações aritméticas)**.**

1. $sum := \lambda m.\lambda n.\lambda fx.mf(nfx)$

2. $mult := \lambda m.\lambda n.\lambda fx.m(nf)x$

3. $suc := \lambda m.\lambda fx.f(mfx)$

Nessas definições os primeiros m e n são os números m e n, como por exemplo $m + n$, $m \times n$, $m + 1$, etc.
Exemplos:

1. Prova que *sum one one* $\twoheadrightarrow_\beta$ *two* na codificação:

$$\begin{aligned} \textit{sum one one} &\equiv (\lambda m.\lambda n.\lambda fx.mf(nfx))\ \textit{one one} \\ &\twoheadrightarrow_\beta (\lambda fx.\textit{one} f(\textit{one} fx)) \\ &\twoheadrightarrow_\beta (\lambda fx.(\lambda gx.gx)f((\lambda gx.gx)fx)) \\ &\twoheadrightarrow_\beta (\lambda fx.(\lambda x.fx)(fx)) \\ &\twoheadrightarrow_\beta (\lambda fx.f(fx)) \\ &\equiv \textit{two} \end{aligned}$$

2. Prova que *mult two two* $\twoheadrightarrow_\beta$ *four* na codificação:

$$\begin{aligned} \textit{mult two two} &\equiv (\lambda m.\lambda n.\lambda fx.m(nf)x)\ \textit{two two} \\ &\twoheadrightarrow_\beta (\lambda fx.\ \textit{two}(\ \textit{two}\ f)x) \\ &\twoheadrightarrow_\beta (\lambda fx.(\lambda gy.g(gy))(\ \textit{two}\ f)x) \end{aligned}$$

Uma vez definida a multiplicação e a soma, é possível definir outras operações como o fatorial e a exponenciação. Isso fica como exercício para o leitor.

Tendo definido operações relacionadas aos números naturais, pode-se perguntar se é possível construir algo lógico dentro do cálculo λ não-tipado. Para isso, é necessário definir a noção de "verdadeiro" e "falso", na forma:

Definição 2.21 (Booleanos)**.**

1. *true* $:= \lambda xy.x$

2. *false* $:= \lambda xy.y$

3. *not* $:= \lambda z.z$ *false true*

4. *'if x then u else v'* $:= \lambda x.xuv$

Exemplos:

1. Prova que $\mathit{not}(\mathit{not}\ p) \equiv p$ na codificação:

$$\begin{aligned}
\mathit{not}(\mathit{not}\ p) &\equiv \mathit{not}((\lambda z.z\ \mathit{false}\ \mathit{true}\ )p) \\
&\twoheadrightarrow_\beta \mathit{not}(p\ \mathit{false}\ \mathit{true}\ ) \\
&\twoheadrightarrow_\beta \mathit{not}(p(\lambda xy.y)(\lambda xy.x)) \\
&\twoheadrightarrow_\beta (\lambda z.z\ \mathit{false}\ \mathit{true}\ )(p(\lambda xy.y)(\lambda xy.x)) \\
&\twoheadrightarrow_\beta (p(\lambda xy.y)(\lambda xy.x))\ \mathit{false}\ \mathit{true}
\end{aligned}$$

   Se $p \twoheadrightarrow_\beta \mathit{true}$,

$$\begin{aligned}
\mathit{not}(\mathit{not}\ \mathit{true}\ ) &\twoheadrightarrow_\beta ((\lambda xy.x)(\lambda xy.y)(\lambda xy.x))\ \mathit{false}\ \mathit{true} \\
&\twoheadrightarrow_\beta ((\lambda xy.y))\ \mathit{false}\ \mathit{true} \\
&\twoheadrightarrow_\beta \mathit{true}
\end{aligned}$$

   Se $p \twoheadrightarrow_\beta \mathit{false}$,

$$\begin{aligned}
\mathit{not}(\mathit{not}\ \mathit{false}\ ) &\twoheadrightarrow_\beta ((\lambda xy.y)(\lambda xy.y)(\lambda xy.x))\ \mathit{false}\ \mathit{true} \\
&\twoheadrightarrow_\beta ((\lambda xy.x))\ \mathit{false}\ \mathit{true} \\
&\twoheadrightarrow_\beta \mathit{false}
\end{aligned}$$

2.2 Modelos

Na matemática, um **modelo** é uma forma de dar sentido à estrutura sintática desenvolvida. No Cálculo λ, os primeiros modelos só foram desenvolvidos posteriormente à sintaxe, pois a simples descrição do cálculo na teoria dos conjuntos gerava inconsistências com os axiomas da teoria dos conjuntos.

2.2.1 Estruturas Aplicativas

Primeiro, antes de definir o que é um modelo, é necessário definir um tipo de estrutura algébrica:

Definição 2.22. Uma *Estrutura Aplicativa* é um par $\langle D, \bullet \rangle$, onde D é um conjunto com ao menos dois elementos, chamado de *domínio* da estrutura, e $\bullet$ é um mapeamento de $\bullet : D \times D \to D$.

Os modelos do Cálculo λ serão estruturas aplicativas acrecidas de propriedades extras. A condição de se ter pelo menos dois elementos em D é importante para evitar modelos triviais.

Seja $\mathcal{M} = \langle D, \bullet \rangle$ uma estrutura aplicativa, escreve-se $a \in \mathcal{M}$ caso $a \in D$.

Definição 2.23. Uma estrutura aplicativa $\mathcal{M} = \langle D, \bullet \rangle$ é *extensional* se para $a, b \in D$, têm-se que $\forall x \in D, a \bullet x = b \bullet x \Rightarrow a = b$. a e b são chamadas de *extensionalmente iguais* e são escritos como $a \sim b$.

Definição 2.24. Seja $\mathcal{M} = \langle D, \bullet \rangle$ uma estrutura aplicativa e seja $n \geq 1$. Uma função $\theta : D^n \to D$ é *representável* se, e somente se, D possui um membro a tal que:

$$(\forall d_1, \ldots, d_n \in D) a \bullet d_1 \bullet d_2 \bullet \cdots \bullet d_n = \theta(d_1, \ldots, d_n)$$

Usando a convenção de associação à esquerda, essa equação é lida como:

$$(\ldots((a \bullet d_1) \bullet d_2) \bullet \cdots \bullet d_n) = \theta(d_1, \ldots, d_n)$$

Cada a é chamado de *representante* de θ. O conjunto de todas as funções representáveis de D^n para D é chamado de $(D^n \to D)_{rep}$.

Definição 2.25. Uma *Algebra Combinatória* é uma estrutura aplicativa $\mathbb{D} = \langle D, \bullet \rangle$, onde dados $k, s \in D$,

1. $(\forall a, b \in D)\ k \bullet a \bullet b = a$
2. $(\forall a, b, c \in D)\ s \bullet a \bullet b \bullet c = a \bullet c \bullet (b \bullet c)$.

Uma Algebra combinatória também é chamada de uma estrutura *combinatorialmente completa*

2.2.2 Modelos interpretativos algébricos

O primeiro tipo de modelo para o Cálculo λ surge através das estruturas aplicativas da seguinte forma:

Definição 2.26. Um *modelo* de $\lambda\beta$ é uma tripla $\mathbb{D} = \langle D, \bullet, [\![\ ]\!] \rangle$, onde $\langle D, \bullet \rangle$ é uma estrutura aplicativa e $[\![\ ]\!]$ é um mapeamento que leva para cada λ-termo M e cada valuação ρ, um membro $[\![M]\!]_\rho$ de D tal que:

1. Para toda variável x, $[\![x]\!]_\rho = \rho(x)$
2. Para todos os termos M e N, $[\![MN]\!]_\rho = [\![M]\!]_\rho \bullet [\![N]\!]_\rho$
3. Para toda variável x, termo M e elemento $d \in D$, $[\![\lambda x.M]\!]_\rho \bullet d = [\![M]\!]_{[d/x]\rho}$
4. Para todo termo M e valuações ρ e σ, $[\![x]\!]_\rho = [\![x]\!]_\sigma$, toda vez que $\rho(x) = \sigma(x)$ para todas as variáveis livres x de M
5. Para todo termo M e todas variáveis x e y, $[\![\lambda x.M]\!]_\rho = [\![\lambda y.[y/x]M]\!]_\rho$, dado que $y \notin FV(M)$.
6. Para todo termo M e N, se para todo $d \in D$ tem-se que $[\![M]\!]_{[d/x]\rho} = [\![N]\!]_{[d/x]\rho}$, então $[\![\lambda x.M]\!]_\rho = [\![\lambda x.N]\!]_\rho$

$[\![M]\!]_\rho$ também pode ser escrito como $[\![M]\!]^{\mathbb{D}}_\rho$ ou simplesmente $[\![M]\!]$, quando já se sabe que a interpretação é independente de ρ.

As condições 1 - 6 imitam o comportamento que um modelo de $\lambda\beta$ precisa ter. A condição 6 fornece a interpretação no modelo da regra ξ. Porém, essas condições não são suficientes para mapear $\lambda\beta\eta$, pois elas não dizem nada sobre a η-conversão. Para isso, é necessário adicionar a seguinte definição:

Definição 2.27. Um modelo de $\lambda\beta\eta$ é um λ-modelo que satisfaz a equação $\lambda x.Mx = M$ para todo termo M e $x \notin FV(M)$.

Dada essa definição, pode-se supor que:

Teorema 2.4. Um λ-modelo $\mathbb{D}$ é extensional se, e somente se, ele é um modelo de $\lambda\beta\eta$.

2.2.3 Modelos livres de Sintaxe

O modelo definido anteriormente não define bem o que a estrutura aplicativa precisa ter como propriedades para ser um λ-modelo, já que se prende à sintaxe dos termos de λβ. Seria interessante definir um modelo onde não fosse necessário definir os termos antes de definir a estrutura aplicativa.

Primeiro, é necessário definir uma propriedade sobre modelos no geral:

Definição 2.28. Seja $\mathbb{D} = \langle D, \bullet, [\![]\!] \rangle$ um λ-modelo. Seja $\sim$ a *equivalência extensional* definida na definição 1.23:

$$a \sim b \iff (\forall d \in D)(a \bullet d = b \bullet d)$$

Para cada $a \in D$, a classe de equivalência extensional $\tilde{a}$ é o conjunto definido por:

$$\tilde{a} = \{b \in D : b \sim a\}$$

Para todo $a \in D$ existem M, x, ρ tais que $[\![\lambda x.M]\!]_\rho \in \tilde{a}$. Por exemplo, sejam $M \equiv ux$ e $\rho = [a/u]\sigma$ para toda valuação σ, então $\rho(u) = a$ e $[\![\lambda x.ux]\!]_\rho$ é equivalente extensionalmente a a, pois:

$$[\![\lambda x.ux]\!]_\rho \bullet d = [\![ux]\!]_{[d/x]\rho} = a \bullet d$$

Definição 2.29. (O mapeamento Λ) Seja $a \in D$ e M, x, ρ tais que $[\![\lambda x.M]\!]_\rho \in \tilde{a}$. Somente um membro de $\tilde{a}$ é igual a $[\![\lambda x.M]\!]_\rho$, esse membro será denominado de $\Lambda(a)$, onde $\Lambda : D \to D$ possui as seguintes propriedades:

1. $\Lambda(a) \sim a$
2. $\Lambda(a) \sim \Lambda(b) \iff \Lambda(a) = \Lambda(b)$
3. $a \sim b \impliedby \Lambda(a) = \Lambda(b)$
4. $\Lambda(\Lambda a) = \Lambda a$
5. Existe $e \in D$ tal que $e \bullet a = \Lambda(a)$ para todo $a \in D$.

Um desses e é o membro em D que corresponde ao numeral de Church 1, pois

$$e = [\![1]\!]_\sigma = [\![\lambda xy.xy]\!]_\sigma$$

e

$$[\![\lambda xy.xy]\!]_\sigma \bullet a = [\![\lambda y.xy]\!]_{[a/x]\sigma} = \Lambda(a)$$

Definição 2.30. (λ-modelos livres de sintaxe) Um λ-modelo livre de sintaxe é uma tripla $\langle D, \bullet, \Lambda \rangle$ onde $\langle D, \bullet \rangle$ é uma estrutura aplicativa e Λ é um mapeamento de D para D, e

1. $\langle D, \bullet \rangle$ é uma algebra combinatória (estrutura aplicativa combinatorialmente completa)
2. Para todo $a \in D$, $\Lambda(a) \sim a$
3. Para todo $a, b \in D$, se $a \sim b$, então $\Lambda(a) = \Lambda(b)$

4. Existe um elemento $e \in D$ tal que para todo $a \in D$, $\Lambda(a) = e \bullet a$

Teorema 2.5. Se $\langle D, \bullet, \Lambda \rangle$ é um λ-modelo livre de sintaxe, então é possível construir um λ-modelo $\langle D, \bullet, [\![]\!] \rangle$ definindo:

1. $[\![x]\!]_\rho = \rho(x)$, se x é uma variável
2. $[\![MN]\!]_\rho = [\![M]\!]_\rho \bullet [\![N]\!]_\rho$
3. $[\![\lambda x.N]\!]_\rho = \Lambda(a)$, onde a é qualquer elemento de D tal que $a \bullet d = [\![N]\!]_{[d/x]\rho}$ para todo $d \in D$.

De forma contrária, se $\langle D, \bullet, [\![]\!] \rangle$ é um λ-modelo então é possível construir um modelo livre de sintaxe $\langle D, \bullet, \Lambda \rangle$ definindo $\Lambda(a) = e \bullet a$, onde $e = [\![\lambda yz.yz]\!]_\rho$ para qualquer valuação ρ.

A existência de Λ pode ser caracterizada por um elemento e da seguinte forma:

Teorema 2.6. Seja $\mathbb{D} = \langle D, \bullet \rangle$ uma estrutura aplicativa tal que $\mathbb{D}$ é combinatorialmente completa e existe um elemento $e \in D$ tal que:

1. para todo $a, b \in D$, $e \bullet a \bullet b = a \bullet b$
2. para todo $a, b \in D$, se $a \sim b$, então $e \bullet a = e \bullet b$.

Então $\langle D, \bullet, \Lambda \rangle$ é um λ-modelo livre de contexto, onde $\Lambda : D \to D$ é definida por $\Lambda(a) = e \bullet a$ para todo $a \in D$.

Uma tripla $\langle D, \bullet, e \rangle$ que satisfa a hipótese do teorema anterior é chamada de λ-modelo frouxo de Scott-Meyer.

2.2.4 Ordens Parciais Completas

O modelo mais conhecido para o Cálculo λ é o Modelo de Dana Scott, o D_∞. O modelo de Dana Scott utiliza a noção de Reticulados (Lattices) Completos, mas é possível fazer uma generalização para Ordens Parciais Completas (CPOs). Alguns modelos do Cálculo λ podem ser descritos mais facilmente por CPOs do que por reticulados.

Para não precisar supor muito, é necessário voltar algumas etapas:

Definição 2.31. Seja P um conjunto. Uma *ordem*, também chamada de *ordem parcial*, em P é uma relação binária $\leq$ em P tal que, para todo $x, y, z, \in P$,

1. (Reflexividade) $x \leq x$
2. (antissimetria) Se $x \leq y$ e $y \leq x$, então $x = y$
3. (Transitividade) Se $x \leq y$ e $y \leq z$, então $x \leq z$

O par $(P, \leq)$ é chamado de *Conjunto ordenado*, ou *Poset* (Do inglês, Partially Ordered set).

Exemplos:

- O conjunto $\mathbb{N}$ dos números naturais, junto com a ordem crescente usual é um poset.

- O conjunto $\{A|A \subseteq X\}$ dos subconjuntos de um conjunto X, escrito como $\mathcal{P}(X)$ e denominado de *Conjunto Potência*, é um poset com ordem dada pela inclusão de subconjuntos $A \subseteq B$. Essa ordem é antissimetrica pois se A e A' são subconjuntos de X onde $A \subseteq A'$ e $A' \subseteq A$, então $A = A'$. Reflexividade e transitividade se seguem da mesma maneira.

Existem várias formas de mapear um conjunto ordenado em outro de forma a manter suas propriedades:

Definição 2.32. Sejam P e Q conjuntos ordenados. Um mapeamento $\phi : P \to Q$ é dito:

1. **preservante de ordem** (também chamado de **monótono**) se $x \leq y$ em P implica em $\phi(x) \leq \phi(y)$ em Q
2. **imersivo de ordem**, escrito como $\phi : P \hookrightarrow Q$, se $x \leq y$ em P se, e somente se, $\phi(x) \leq \phi(y)$ em Q
3. **isomorfismo de ordem** se é uma imersão de ordem que mapeia P em Q

Alguns conjuntos possuem um valor menor possível ou um valor maior possível, definidos da seguinte forma:

Definição 2.33. Seja P um conjunto ordenado. P possui um elemento *mínimo* se existe $\bot \in P$ tal que $\bot \leq x$ para todo $x \in P$. De forma dual, P possui um elemento *máximo* $\top \in P$ tal que $x \leq \top$ para todo $x \in P$.

Exemplos:

- O mínimo do conjunto ordenado $(\mathbb{N}, \leq)$ é o 0, mas não existe máximo.
- No conjunto ordenado $(P(X), \subseteq)$, tem-se que $\bot = \emptyset$ e $\top = X$.

Subconjuntos de conjuntos ordenados também podem possuir elementos mínimos e máximos:

Definição 2.34. Seja P um conjunto ordenado e $Q \subseteq P$. Então o elemento $u \in P$ tal que $x \leq u$ para todo $x \in Q$ é chamado de *cota superior* de Q. O elemento $l \in P$ é chamado de *menor cota superior* ou *supremo* de Q se para toda cota superior $u \in P$, $l \leq u$.
Dualmente, o elemento $u \in P$ tal que $u \leq x$ para todo $x \in Q$ é chamado de *cota inferior* de Q. O elemento $l \in P$ é chamado de *maior cota inferior* ou *infimo* de Q se para toda cota inferior $u \in P$, $u \leq l$.

Exemplo: Seja $S = \{1, 3, 5\} \subset \mathbb{N}$, então são cotas inferiores 0 e 1 e são cotas superiores todo número maior que 5.

Supremos e ínfimos podem ser tratados algebricamente da seguinte forma:

Definição 2.35.

1. O *Join* de x e y, $x \vee y$, é o supremo $\sup\{x, y\}$. O supremo de um conjunto qualquer é denotado por $\bigvee S$
2. O *meet* de x e y, $x \wedge y$ é o infimo $\inf\{x, y\}$. O infimo de um conjunto qualquer S é denotado por $\bigwedge S$.

Um reticulado pode ser definido por:

Definição 2.36. (Reticulado) Seja P um conjunto ordenado não vazio, então:

1. Se $x \vee y$ e $x \wedge y$ existem para todo $x, y \in P$, então P é chamado de *Reticulado*
2. Se $\bigvee S$ e $\bigwedge S$ existem para todo $S \subseteq P$, então P é chamado de *Reticulado Completo*

Definição 2.37. Um subconjunto X de P é dito *direcionado* se, e somente se, X é não vazio e para cada par de elementos $x, y \in X$, existe um elemento $z \in X$ tal que $x \leq z$ e $y \leq z$.

Agora finalmente a definição de uma ordem parcialmente completa:

Definição 2.38. Uma *Ordem Parcialmente Completa* (CPO) é um conjunto ordenado parcial $(D, \leq)$ tal que:

1. D possui um elemento mínimo
2. Todo subconjunto direcionado X de D possui um supremo. Ou seja $\bigvee X$ existe para todo $X \subseteq D$

Dessa forma, é possível ver em que medida um CPO é mais geral que um reticulado, pois ele retira a condição que o ínfimo exista para todo $X \subseteq D$.

Exemplo: Seja um objeto $\perp \notin \mathbb{N}$ e seja $\mathbb{N}^+ = \mathbb{N} \cup \{\perp\}$. Defina um ordenamento em $\mathbb{N}^+$ como:

$$a \sqsubseteq b \text{ sse } (a = \perp \text{ e } b \in \mathbb{N}) \text{ ou } a = b$$

. O par $(\mathbb{N}^+, \sqsubseteq)$ é um CPO.

É possível descrever *morfismos* entre CPOs:

Definição 2.39. Sejam D e D' cpos e $\phi : D \to D'$ uma função,

1. ϕ é chamada *monotônica* sse $a \leq b$ implica em $\phi(a) \leq' \phi(b)$
2. ϕ é chamada *contínua* sse para todo subconjunto direcionado X de D, $\phi(\bigvee X) = \bigvee \phi(X)$.

O conjunto de todas as funções contínuas entre D e D' é denotado por $[D \to D']$.

Em $[D \to D']$ é possível definir uma relação $\preceq$ tal que:

$$\phi \preceq \psi \leftrightarrow \phi(d) \leq' \psi(d) \text{ para todo } d \in D$$

Então $\preceq$ é uma ordem parcial em $[D \to D']$ e $[D \to D']$ possui um elemento final:

$$\perp(d) = \perp' \text{ para todo } d \in D$$

E, se Φ é um subconjunto direcionado de $[D \to D']$, entã opara todo $d \in D$ o conjunto $\{\phi(d) | \phi \in \Phi\}$ é um subconjunto direcionado de D'. Com isso, é possível definir uma função $\psi : D \to D'$ como:

$$\psi(d) = \bigvee \{\phi(d) | \phi \in \Phi\} \text{ para todo } d \in D$$

Então, é possível monstrar a seguinte proposição:

Proposição 2.1. Se D e D′ são cpos, então $[D \to D']$ também é um cpo pelo ordenamento parcial $\preceq$ definido anteriormente. Seu último elemento é dado por $\perp'$ e para qualquer subconjunto direcionado Φ de $[D \to D']$, $\bigvee \Phi$ é uma função ψ definida como anteriormente.

Dado um cpo D_0, pode-se construir uma sequência $\{D_n\}_{n=0}^{\infty}$ de cpos definidos indutivamente como $D_{n+1} = [D_n \to D_n]$ para todo $n \geq 0$. O modelo D_∞ de Scott parte de $D_0 = \mathbb{N}^+$

Para estudar a relação das sequências $\{D_n\}_{n=0}^{\infty}$ entre si, é interessante pensar a relação de como um cpo pode estar *mergulhado* em outro.

Definição 2.40. Sejam D e D′ cpos. Uma *projeção* de D em D′ é um par $\langle \phi, \psi \rangle$ de funções com $\phi \in [D \to D']$ e $\psi \in [D' \to D]$ tais que:

$$\psi \circ \phi = I_D \text{ e } \phi \circ \psi \preceq I'_D$$

Onde I_D e I'_D são as funções identidade em D e D′ respectivamente.

Se $\langle \phi, \psi \rangle$ é uma projeção de D′ em D então ϕ mergulha D em D′.

Para entender a composição de ϕ e ψ é necessário definir o seguinte lema:

Lema 2.6. A composição de funções contínuas entre cpos é contínua. Ou seja, se D, D′ e D″ são cpos e $\psi \in [D \to D']$ e $\phi \in [D' \to D'']$ e $\phi \circ \psi$ é definido por

$$\text{para todo } d \in D (\phi \circ \psi)(d) = \phi(\psi(d))$$

Então

$$\phi \circ \psi \in [D \to D'']$$

Usando a definição de $\{D_n\}_{n=0}^{\infty}$ é possível construir uma projeção $\langle \phi_n, \psi_n \rangle$ de D_{n+1} para D_n para cada n. A projeção inicial de D_1 em D_0 pode ser montada da seguinte forma: Para cada $d \in D_0$, seja κ_d uma função constante $\kappa_d(c) = d$ para $c \in D_0$. κ_d é contínua, então $\kappa_d \in [D_0 \to D_0] = D_1$. Seja $\phi_0 : D_0 \to D_1$ e $\psi_0 : D_1 \to D_0$ tais que $\phi_0(d) = \kappa_d$ para $d \in D_0$ e $\psi_0(c) = c(\perp_0)$ para $c \in D_1$ (onde $\perp_0$ é o menor elemento de D_0). ϕ_0 e ψ_0 são contínuas e é possível ver que

$$(\psi_0 \circ \phi_0)(d) = \kappa_d(\perp_0) = d = I_{D_0}$$

e

$$(\phi_0 \circ \psi_0)(f) = \kappa_d(f(\perp_0)) \preceq I_{D_1}$$

Logo $\langle \phi_0, \psi_0 \rangle$ é uma projeção de D_1 em D_0. Agora seja $\phi_n : D_n \to D_{n+1}$ e $\psi_n : D_{n+1} \to D_n$ gerados indutivamente por:

$$\phi_n(\sigma) = \phi_{n-1} \circ \sigma \circ \psi_{n-1} \text{ e } \psi_n(\tau) = \psi_{n-1} \circ \tau \circ \phi_{n-1}$$

para $\sigma \in D_n$ e $\tau \in D_{n+1}$. É possível monstrar que $\phi_n \in [D_n \to D_{n+1}]$ e $\psi_n \in [D_{n+1} \to D_n]$. Logo o par $\langle \phi_n, \psi_n \rangle$ é uma projeção de D_{n+1} em D_n.

As funções ψ_n e ϕ_n só elevam n um número por vez, então é possível definir uma função $\phi_{m,n}$ da seguinte forma:

Definição 2.41. Para qualquer $m, n \geq 0$, $\phi_{m,n} : D_m \to D_n$ é definido da seguinte forma:

$$\phi_{m,n} = \begin{cases} \phi_{n-1} \circ \phi_{n-2} \circ \cdots \circ \phi_{m+1} \circ \phi_m & \text{se } m < n \\ I_{D_n} & \text{se } m = n \\ \psi_n \circ \psi_{n+1} \circ \cdots \circ \psi_{m-2} \circ \psi_{m-1} & \text{se } m > n \end{cases}$$

Uma vez definida essa função, é possível monstrar o seguinte lema:

Lema 2.7. Sejam $m, n \geq 0$, então:

1. $\phi_{m,n} \in [D_m \to D_n]$
2. Se $m \leq n$, então $\phi_{n,m} \circ \phi_{m,n} = I_{D_m}$
3. Se $m > n$, então $\phi_{n,m} \circ \phi_{m,n} \preceq I_{D_m}$
4. Se k é um número entre m e n, então $\phi_{k,n} \circ \phi_{m,k} = \phi_{m,n}$

Prova:

1. Em $\phi_{m,n}$ existem três casos:
   (a) Se $n > m$, então $\phi_{m,n} = \phi_{n-1} \circ \phi_{n-2} \circ \cdots \circ \phi_{m+1} \circ \phi_m$. É facil ver que, pelo lema da composição, sendo $\phi_m \in [D_m \to D_{m+1}]$ o inicio da cadeia de composições e $\phi_{n-1} \in [D_{n-1} \to D_n]$ o fim dessas cadeia, e sendo essa cadeia de composições contínua, então $\phi_{m,n} \in [D_m \to D_n]$
   (b) Se $n = m$, $\phi_{n,n} = I_{D_n}$, mas $I_{D_n} \in [D_n \to D_n]$, logo $\phi_{n,n} \in [D_n \to D_n]$
   (c) Se segue de forma análoga a (a)
2. Existem dois casos:
   (a) Se $m = n$, $\phi_{n,n} \circ \phi_{n,n} = I_{D_n}$
   (b) Se $m < n$, $\phi_{m,n} \in [D_m \to D_n]$ e $\phi_{n,m} \in [D_n \to D_m]$. Pelo lema da composição, $\phi_{n,m} \circ \phi_{m,n} \in [D_n \to D_n]$. O valor de $\phi_{n,m} \circ \phi_{m,n} \preceq I_{D_m}$ se segue da definição de projeção.
3. Deixado para o leitor
4. Deixado para o leitor

2.2.5 O Modelo de Scott

Uma vez feitas essas definições sobre cpos, é possível definir o modelo de Scott. Para isso, é necessário definir D_∞:

Definição 2.42.

- D_∞ é o conjunto de todas as sequências infinitas na forma

  $$d = \langle d_0, d_1, d_2, \dots \rangle$$

  tais que para todo $n \geq 0$ tem-se $d_n \in D_n$ e $\psi_n(d_{n+1}) = d_n$
- A relação $\sqsubseteq$ em D_∞ possui a forma:

  $$d = \langle d_0, d_1, d_2, \dots \rangle \sqsubseteq \langle d'_0, d'_1, d'_2, \dots \rangle \text{ se } d_n \sqsubseteq d'_n \text{ para todo } n \geq 0$$

- Se X é um subconjunto de D_∞, então $X_n = \{a_n | a \in X\}$ é o conjunto dos n-ésimos termos de cada sequência $a \in X$.

Lema 2.8. O par $\langle D_\infty, \sqsubseteq \rangle$ definido acima é um cpo com menor elemento

$$\bot = \langle \bot_0, \bot_1, \bot_2, \dots \rangle$$

onde $\bot_n$ é o menor elemento de D_n e menor cota superior do subconjunto direcionado X de D_∞ dado por:

$$\bigvee X = \langle \bigvee X_0, \bigvee X_1, \bigvee X_2, \dots \rangle$$

Para cada $n \geq 0$ é possível definir um par de funções contínuas que formam uma projeção de D_∞ em D_n definidas como:

Definição 2.43. Para cada $n \geq 0$, seja $\phi_{n,\infty} : D_\infty \to D_n$ e $\phi_{\infty,n} : D_n \to D_\infty$ definidas por:

$$\phi_{n,\infty} = \langle \phi_{n,0}(d), \phi_{n,1}(d), \phi_{n,2}(d), \dots \rangle$$

para todo $d \in D_n$ e

$$\phi_{\infty,n}(d) = d_n$$

para todo $d \in D_\infty$

Lema 2.9. Sejam $m, n \geq 0$ com $m \leq n$ e $a, b \in D_\infty$, então

1. O par $\langle \phi_{n,\infty}, \phi_{\infty,n} \rangle$ é uma projeção de D_∞ em D_n
2. $\phi_{m,n}(a_m) \sqsubseteq a_n$
3. $\phi_{m,\infty}(a_m) \sqsubseteq \phi_{n,\infty}(a_n)$
4. $a = \bigvee_{n \geq 0} \phi_{n,\infty}(a_n)$
5. $\phi_{n,\infty}(a_{n+1}(b_n)) \sqsubseteq \phi_{n+1,\infty}(a_{n+2}(b_{n+1}))$

A parte 4 sugere que os termos a_n servem como aproximações cada vez mais certas de a em D_∞. Com isso, é possível ver a aplicação $(a_{n+1}(b_n))$ quando $n \to \infty$ como uma aproximação cada vez melhor da aplicação ab. Logo, é possível definir uma relação binária em D_∞ da seguinte forma:

Definição 2.44. Para todo $a, b \in D_\infty$,

$$a \bullet b = \bigvee \{\phi_{n,\infty}(a_{n+1}(b_n)) | n \geq 0\}$$

A autoaplicação presente no Cálculo λ pode ser implementada utilizando a aplicação $a_{n+1}(a_n)$.

Uma vez definida a relação binária, pode-se ver que o par $\langle D_\infty, \bullet \rangle$ é uma estrutura aplicativa. Pode-se definir um modelo livre de sintaxe a partir dessa estrutura. Para isso, é necessário mostrar que o par $\langle D_\infty, \bullet \rangle$ é uma algebra combinatória, ou seja mostrar que existem k e s que satisfaçam as condições da Definição 1.25.

Definição 2.45 (k_n, s_n)**.**

1. Seja $n \geq 2$. Para $a \in D_{n-1}$, $\kappa_a : D_{n-2} \to D_{n-2}$ é a função constante $\kappa_a = \psi_{n-2}(a)$ para todo $b \in D_{n-2}$. Então $k_n : D_{n-1} \to D_{n-1}$ é $k_n(a) = \kappa_a$ para todo $a \in D_{n-1}$.

2. Seja $n \geq 3$. Para $a \in D_{n-1}$ e $a \in D_{n-2}$, $\tau_{a,b} : D_{n-3} \to D_{n-3}$ é a função constante $\tau_{a,b} = a(\phi_{n-3}(c))(b(c))$ para todo $c \in D_{n-3}$ e $\sigma_a : D_{n-2} \to D_{n-2}$ tal que $\sigma_a = \tau_{a,b}$ para todo $b \in D_{n-2}$. Então $s_n : D_{n-1} \to D_{n-1}$ é $s_n(a) = \sigma_a$ para todo $a \in D_{n-1}$.

Lema 2.10.

1. Para todo $n \geq 2$, tem-se que $k_n \in D_n$ e $\psi_n(k_{n+1}) = k_n$. Logo $\psi_1(k_2) = I_{D_0} \in D_1$.
2. Para todo $n \geq 3$, tem-se que $s_n \in D_n$ e $\psi_n(s_{n+1}) = s_n$. Logo $\psi_1(\psi_2(s_3)) = I_{D_0} \in D_1$.

Agora finalmente pode-se definir k e s:

Definição 2.46. Sejam k e s as seguintes sequências:

$$k = \langle \perp_0, I_{D_0}, k_2, k_3, \ldots \rangle \text{ e } s = \langle \perp_0, I_{D_0}, \psi_2(s_3), k_3, k_4, \ldots \rangle$$

Lema 2.11. As sequências k e s são elementos de D_∞

Lema 2.12. Para todo $a, b, c \in D_\infty$, $k \bullet a \bullet b = a$ e $s \bullet a \bullet b \bullet c = a \bullet c \bullet (b \bullet c)$

Logo, pelo lema anterior, é possível ver que o par $\langle D_\infty, \bullet \rangle$ é uma álgebra combinatória. O que falta para provar que esse par é um modelo é monstrar que ele é extensional.

Lema 2.13. $\langle D_\infty, \bullet \rangle$ é uma álgebra combinatória extensional

Prova: Sejam a e b elementos de D_∞ tais que $a \sim b$, ou seja, $a \bullet c = b \bullet c$ para todo $c \in D_\infty$. Seja $m \geq 0$ e d um elemento arbitrário de D_m. Seja $c = \phi_{m,\infty}(d)$. Pode ser provado que $(a \bullet c)_m = a_{m+1}(d)$ e $(b \bullet c)_m = b_{m+1}(d)$. Logo $a_{m+1} = (a \bullet c)_m = (b \bullet c)_m(d) = b_{m+1}$ e $a_{m+1} = b_{m+1}$. Ou seja $a_n = b_n$ para $n > 0$ e $a_0 = \psi_0(a_1) = \psi_0(b_1) = b_0$ (Pois ψ é contínua), logo $a = b$. Logo, $\langle D_\infty, \bullet \rangle$ é extensional.

Com isso, fica provado que $\langle D_\infty, \bullet \rangle$ é um λ-modelo livre de sintaxe

3 Teoria dos Tipos Simples

O cálculo λ não-tipado possui alguns entraves ao tentar traduzir as funções matemáticas para seus termos. Um desses entraves é o fato que as funções matemáticas são mapeamentos entre dois conjuntos. Ou seja, essas funções possuem em sua definição os valores que vão esperar e os possíveis valores que vão retornar. A função soma $+ : \mathbb{N} \to \mathbb{N}$ não pode aceitar os valores $\mathtt{true}$ ou $\mathtt{false}$. Porém, nas codificações do cálculo λ descrito até então (Sem contar com os modelos), isso é possível. Por exemplo, é possível perceber que $\mathtt{false}$ e $\mathbb{0}$ são definidos pelo mesmo termo $\lambda xy.y$ (a definição de $\mathbb{0}$ é α-equivalente a essa), o que pode gerar confusão em sua aplicação.

Outro problema do Cálculo λ não-tipado é o fato de poder existir recursões infinitas através de termos como Ω e Δ. A tipagem dos termos faz com que esse tipo de fenômeno não ocorra. O que retira a Turing-completude, mas facilita outras coisas.

Para fazer essa descrição ser mais detalhada e evitar esse tipo de erro, Church introduziu tipos.

3.1 Cálculo λ simplesmente tipado (STλC)

3.1.1 Tipos simples

Uma forma simples de começar a tipagem dos λ-termos é considerando uma coleção de variáveis de tipos e uma forma de produzir mais tipos através dessa coleção, chamado de *tipo funcional*

Seja $\mathbb{V}$ a coleção infinita de variáveis de tipos $\mathbb{V} = \{\alpha, \beta, \gamma, \dots\}$, então:

Definição 3.1 (A coleção de todos os tipos simples)**.** A coleção dos tipos simples $\mathbb{T}$ é definida por:

1. (Variável de tipos) Se $\alpha \in \mathbb{V}$, então $\alpha \in \mathbb{T}$
2. (Tipo funcional) Se $\sigma, \tau \in \mathbb{T}$, então $(\sigma \to \tau) \in \mathbb{T}$.

Na BNF, $\mathbb{T} = \mathbb{V} | \mathbb{T} \to \mathbb{T}$

Os parenteses no tipo funcional são associativos à direita, ou seja o tipo $\alpha_1 \to \alpha_2 \to \alpha_3 \to \alpha_4$ é $(\alpha_1 \to (\alpha_2 \to (\alpha_3 \to \alpha_4)))$

Tipos simples arbitrários serão escritos com letras gregas minúsculas (Com excessão do λ) como $\sigma, \tau, \dots$, mas também podem ser escrito como letras latinas maiúsculas $A, B, \dots$ na literatura.

As variáveis de tipos são representações abstratas de tipos básicos como os números naturais $\mathbb{N}$ ou a coleção de todas as listas $\mathbb{L}$. Esses tipos serão explorados mais à frente. Já os tipos funcionais representam funções na matemática como por exemplo $\mathbb{N} \to \mathbb{N}$, o conjunto de funções que leva dos naturais para os naturais, ou $(\mathbb{N} \to \mathbb{Z}) \to \mathbb{Z} \to \mathbb{N}$, o conjunto de funções que recebem como entrada uma função que leva dos naturais aos inteiros e um inteiro e retorna um natural.

A sentença "O termo M possui tipo σ" é escrita na forma $M : \sigma$. Todo termo possui um tipo único, logo se x é um termo e $x : \sigma$ e $x : \tau$, então $\sigma \equiv \tau$.

Como os tipos foram introduzidos para lidar com o cálculo λ, eles devem ter regras para lidar com as operações de aplicação e abstração.

1. (*Aplicação*): No cálculo λ, sejam M e N termos, podemos fazer uma aplicação entre eles no estilo MN. Para entender como entram os tipos, é possível recordar de onde surge a intuição para a aplicação. Seja $f : \mathbb{N} \to \mathbb{N}$ a função $f(x) = x^2$, então, a aplicação de 3 em f é $f(3) = 3^2$. Nesse exemplo, omite-se o fato que para aplicar 3 a f, 3 tem que estar no domínio de f, ou seja, $3 \in \mathbb{N}$. No caso do cálculo λ, para aplicar N em M, M deve ter um tipo funcional, na forma $M : \sigma \to \tau$, e N deve ter como tipo o primeiro tipo que aparece em M, ou seja $N : \sigma$.

2. (*Abstração*): No cálculo λ, seja M um termo, podemos escrever um termo $\lambda x.M$. A abstração "constroi" a função. Para a tipagem, seja $M : \tau$ e $x : \sigma$, então $\lambda x : \sigma.M : \sigma \to \tau$. É possível omitir o tipo da variável, escrevendo no estilo: $\lambda x.M : \sigma \to \tau$.

Alguns exemplos:

1. Seja x do tipo σ, a função identidade é escrita na forma $\lambda x.x : \sigma \to \sigma$.

2. O combinador $\mathbf{B} \equiv \lambda xyz.x(yz)$ é tipado na forma $\mathbf{B} : (\sigma \to \tau) \to (\rho \to \sigma) \to \rho \to \tau$.

3. O combinador $\Delta \equiv \lambda x.xxx$ não possui tipagem. Isso ocorre pois, na aplicação xx, x precisa ter como tipo $\sigma \to \tau$ e σ, mas como x só pode ter um tipo, então $\sigma \to \tau \equiv \sigma$. O que não é possível em $\mathbb{T}$. Logo Δ (e Ω por motivos similares), não faz parte da teoria dos tipos simples.

O último exemplo mostra que o teorema do ponto fixo não ocorre para todos os termos na teoria dos tipos simples e que não existe recursão infinita, fazendo com que a teoria dos tipos simples deixe de ser turing-completa.

3.1.2 Abordagens para a tipagem

Existem duas formas de tipar um λ-termo:

1. (*Tipagem à la Church / Tipagem explícita / Tipagem intrínseca / Tipagem ontológica*) Nesse estilo de tipagem, só termos que possuem tipagem que satisfaz a construção de tipos interna à teoria são aceitos. Cada termo possui um tipo único.

2. (*Tipagem à la Curry / Tipagem implícita / Tipagem extrínseca / Tipagem semântica*) Nesse estilo de tipagem, os termos são os mesmos do cálculo λ não tipado e pode-se não definir o tipo do termo na sua introdução, mas deixá-lo aberto. Os tipos são buscados para o termo, por tentativa e erro.

Exemplos

1. (Tipagem intrínseca): Seja x do tipo $\alpha \to \alpha$ e y do tipo $(\alpha \to \alpha) \to \beta$, então yx possui o tipo β. Se z possuit tipo β e u possuir tipo γ, então $\lambda zu.z$ tem tipo $\beta \to \gamma \to \beta$ e a aplicação $(\lambda zu.z)(yx)$ é permitida pois o tipo β de yx equivale ao tipo β que $\lambda zu.z$ recebe.

2. (Tipagem extrínseca): Nessa tipagem, começa-se com o termo $M \equiv (\lambda zu.z)(yx)$ e tenta-se adivinhar qual seu tipo e o tipo de suas variáveis. É possível notar que $(\lambda zu.z)(yx)$ é uma aplicação, então $(\lambda zu.z)$ precisa ter um tipo $A \to B$, yx precisa ter um tipo A e M terá um tipo B. Mas se $\lambda zu.z$ possui o tipo $A \to B$, então $\lambda u.z$ possui o tipo B e, como o termo é uma abstração, B precisa ser um tipo funcional, ou seja $B \equiv C \to D$. Logo $u : C$ e $z : D$. Já no caso de $yx : A$, y precisa ter um tipo funcional para ser aplicado a x, logo sendo $x : E$, $y : E \to F$. Logo temos que $x : E, y : E \to A, z : A, u : C$. Só é necessário então substituir A, C, E com tipos variáveis como α, β, γ: $x : \alpha, y : \alpha \to \beta, z : \beta, u : \gamma$.

No caso do exemplo 2, é possível escrever $x : \alpha, y : \alpha \to \beta, z : \beta, u : \gamma \vdash (\lambda z : \beta.\lambda u : \gamma.z)(yx) : \gamma \to \beta$. A lista à esquerda da $\vdash$ (lê-se catraca) é chamada de *contexto*.

3.1.3 Regras de derivação e Cálculo de sequêntes

É necessário, na tipagem intrínseca, definir a coleção de todos os λ-termos tipados:

Definição 3.2 (λ-termos pré-tipados)**.** A coleção $\Lambda_{\mathbb{T}}$ de λ-termos pré-tipados é definida pela BNF:
$\Lambda_{\mathbb{T}} = V|(\Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}})|(\lambda V : \mathbb{T}.\Lambda_{\mathbb{T}})$

Para expressar as tipagens dos λ-termos, é necessário desenvolver um conjunto de definições que ainda não foram mostradas:

Definição 3.3.

1. Uma *sentença* é $M : \sigma$, onde $M \in \Lambda_{\mathbb{T}}$ e $\sigma \in \mathbb{T}$. Nessa sentença, M é chamado de *sujeito* e σ de *tipo*
2. Uma *declaração* é uma sentença com uma *variável* como sujeito
3. Um *Contexto* é uma lista, possivelmente nula, de declarações com diferentes sujeitos
4. Um *Juizo* possui a forma $\Gamma \vdash M : \sigma$, onde Γ é o contexto e $M : \sigma$ é uma sentença.

Para estudar a tipagem, será utilizado um sistema de derivações trazido da lógica chamado de *Cálculo de sequêntes*. O cálculo de sequêntes dá a possibilidade de gerar juizos de forma formal utilizando árvores de derivação no estilo:

$$\frac{\text{premissa 1} \qquad \text{premissa 2} \qquad \dots \qquad \text{premissa } n}{\text{Conclusão}}$$

Acima da linha horizontal estão as premissas, que são cada uma um juizo, e abaixo da linha horizontal está a conclusão, que é em si um juizo também. A linha marca uma regra de derivação específica da teoria que se está trabalhando.

Definição 3.4 (Regras de derivação para o STλC)**.**

- (*var*) $\Gamma \vdash x : \sigma$, dado que $x : \sigma \in \Gamma$.
- (*appl*)

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{appl}$$

- (*abst*)

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau}\ \text{abst}$$

A regra *(var)* não possui premissas e possui como conclusão o fato que dado um contexto Γ, se existe uma declaração em Γ, essa declaração é derivável através de Γ. Essa primeira regra é tratada como axioma em (Hindley, 1997), pois, assim como todo axioma, ela é derivável sem precisar de premissas. Na construção da árvore de dedução, essa regra está no topo como uma "raiz".

A regra *(appl)* é equivalente no cálculo ao que foi feito antes. Essa regra também é chamada na literatura de $\to -\mathtt{elim}$ ou $\to \mathrm{E}$

A regra *(abs)* é equivalente no cálculo à abstração e pode ser chamada na literatura de $\to -\mathtt{intro}$ ou $\to \mathrm{I}$.

Exemplo:

$$\dfrac{\dfrac{\dfrac{(1)\ y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta \qquad (2)\ y : \alpha \to \beta, z : \alpha \vdash z : \beta}{(3)\ y : \alpha \to \beta, z : \alpha \vdash yz : \beta}\ \text{appl}}{(4)\ y : \alpha \to \beta \vdash \lambda z : \alpha.yz : \alpha \to \beta}\ \text{abs}}{(5)\ \emptyset \vdash \lambda y : \alpha \to \beta.\lambda z : \alpha.yz : (\alpha \to \beta) \to \alpha \to \beta}\ \text{abs}$$

Dada a derivação já montada, sua leitura pode ser feita de baixo para cima, feito levando em conta as premissas mais fundamentais até a conclusão final, de forma a adicionar informação aos juízos a cada passo, ou de cima para baixo, feito para entender qual caminho leva até o objetivo final.

1. Os passos (1) e (2) usam a regra (*var*)
2. O passo (3) usa a regra (*app*) usando (1) e (2) como premissas
3. O passo (4) usa a regra ((abs)) com (3) como premissa
4. O passo (5) usa a regra ((abs)) com (4) como premissa

As regras de derivação podem ser entendidas em outros contextos:

Matemática: Seja $A \to B$ o conjunto de todas as funções de A para B, então as regras se tornam:

1. (*aplicação funcional*)

$$\frac{\text{se } f \text{ é um membro de } A \to B \qquad \text{e se } c \in A}{\text{então } f(c) \in B}$$

2. (*abstração funcional*)

$$\frac{\text{Se para } x \in A \text{ segue-se que } f(x) \in B}{\text{então } f \text{ é membro de } A \to B}$$

Lógica: Seja $A \Rightarrow B$ "A implica em B", então pode-se ler $A \to B$ como $A \Rightarrow B$. As regras se tornam:

1. $(\Rightarrow -\mathtt{elim})$

$$\frac{A \to B \qquad A}{B}$$

2. $(\Rightarrow -\mathtt{intro})$

$$\begin{array}{c} A \\ \vdots \\ \hline B \end{array}$$

A regra de eliminação é denominada de *Modus Ponens*. Ambas as regras como estão escritas aí são parte das regras definidas na *Dedução Natural*, um cálculo análogo ao cálculo de sequêntes (Toda árvore definida na dedução natural possui um equivalente no cálculo de sequêntes). Esse estilo de dedução natural é chamado de *Dedução natural no estilo de Gentzen*, para diferenciá-lo da *Dedução natural no estilo de Fitch* que é escrito como:

Definição 3.5 ($\lambda_{\to}$-termos legais)**.** Um termo M pré-tipado em $\lambda_{\to}$ é chamado *legal* se existe um contexto Γ e um tipo ρ tal que $\Gamma \vdash M : \rho$.

3.1.4 Problemas resolvidos no STLC

No geral, existem três tipos de problemas relacionados a julgamentos na teoria dos tipos:

1. *Bem-tipagem* (*Well-typedness*) ou *Tipabilidade*: esse problema surge da questão

$$? \vdash \text{termo} : ?$$

Ou seja, saber se um termo é legal e, se não é, mostrar onde sua contrução falha.
(1a) *Atribuição de tipos*, que surge da questão:

$$\text{contexto} \vdash \text{termo} : ?$$

. Ou seja, dado um contexto e um termo, derive seu tipo.

2. *Checagem de tipos*, que surge da questão

$$\text{contexto} \vdash^{?} \text{termo} : \text{tipo}$$

. Ou seja, se é realmente verdadeiro que o termo possui o tipo no determinado contexto.

3. *Encontrar o termo*, que surge da questão:

$$\text{contexto} \vdash ? : \text{tipo}$$

. Um tipo particular desse problema é quando o contexto é vazio, ou seja

$$\emptyset \vdash ? : \text{tipo}$$

.

Todos esses problemas são *decidíveis* em $\lambda_{\rightarrow}$. Ou seja, para cada um deles existe um *algoritmo* (um conjunto de passos) que produz a resposta. Em outros sistemas, encontar um termo se torna *indecidível*.

3.1.5 Bem-tipagem em $\lambda_{\rightarrow}$

Para exemplificar os passos necessários para resolver a bem-tipagem em $\lambda_{\rightarrow}$, será utilizado o exemplo descrito em 1.1.3, dessa vez passo a passo.

O objetivo é mostrar que o termo $M \equiv \lambda y : \alpha \rightarrow \beta.\lambda z : \alpha.yz$ é um termo legal. Logo, precisamos encontrar um contexto Γ e um tipo ρ tal que $\Gamma \vdash M : \rho$.

Primeiro, como não existem variáveis livres em M, o contexto inicial pode ser considerado vazio: $\Gamma = \emptyset$.

Inicialmente, o primeiro passo é descobrir qual a premissa, ou premissas, que gera o termo e a regra de dedução:

$$\frac{?}{\emptyset \vdash \lambda y : \alpha \rightarrow \beta.\lambda z : \alpha.yz : \ldots}\ ?$$

Como a primeira parte do termo é um λy, a única regra possível inicialmente é a abstração:

$$\frac{\dfrac{?}{y : \alpha \rightarrow \beta \vdash \lambda z : \alpha.yz : \ldots}\ ?}{\emptyset \vdash \lambda y : \alpha \rightarrow \beta.\lambda z : \alpha.yz : \ldots}\ \text{abs}$$

Novamente, a única regra possível é a abstração:

$$\frac{\dfrac{\dfrac{?}{y : \alpha \rightarrow \beta, z : \alpha \vdash yz : \ldots}\ ?}{y : \alpha \rightarrow \beta \vdash \lambda z : \alpha.yz : \ldots}\ \text{abs}}{\emptyset \vdash \lambda y : \alpha \rightarrow \beta.\lambda z : \alpha.yz : \ldots}\ \text{abs}$$

Sobrou do lado direito da catraca o termo yz que, vendo o contexto, é a aplicação de outros dois termos, logo a única regra possível é a aplicação:

$$\frac{\dfrac{\dfrac{y : \alpha \rightarrow \beta, z : \alpha \vdash y : \alpha \rightarrow \beta \qquad y : \alpha \rightarrow \beta, z : \alpha \vdash z : \beta}{y : \alpha \rightarrow \beta, z : \alpha \vdash yz : \ldots}\ \text{appl}}{y : \alpha \rightarrow \beta \vdash \lambda z : \alpha.yz : \ldots}\ \text{abs}}{\emptyset \vdash \lambda y : \alpha \rightarrow \beta.\lambda z : \alpha.yz : \ldots}\ \text{abs}$$

Como as premissas mais superiores são geradas de (*var*), não há mais nenhum passo de premissas e a tipagem pode ser realizada de cima para baixo.

$$\dfrac{\dfrac{\dfrac{y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta \qquad y : \alpha \to \beta, z : \alpha \vdash z : \beta}{y : \alpha \to \beta, z : \alpha \vdash yz : \beta}\ \text{appl}}{y : \alpha \to \beta \vdash \lambda z : \alpha . yz : \ldots}\ \text{abs}}{\emptyset \vdash \lambda y : \alpha \to \beta . \lambda z : \alpha . yz : \ldots}\ \text{abs}$$

$$\dfrac{\dfrac{\dfrac{y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta \qquad y : \alpha \to \beta, z : \alpha \vdash z : \beta}{y : \alpha \to \beta, z : \alpha \vdash yz : \beta}\ \text{appl}}{y : \alpha \to \beta \vdash \lambda z : \alpha . yz : \alpha \to \beta}\ \text{abs}}{\emptyset \vdash \lambda y : \alpha \to \beta . \lambda z : \alpha . yz : \ldots}\ \text{abs}$$

$$\dfrac{\dfrac{\dfrac{y : \alpha \to \beta, z : \alpha \vdash y : \alpha \to \beta \qquad y : \alpha \to \beta, z : \alpha \vdash z : \beta}{y : \alpha \to \beta, z : \alpha \vdash yz : \beta}\ \text{appl}}{y : \alpha \to \beta \vdash \lambda z : \alpha . yz : \alpha \to \beta}\ \text{abs}}{\emptyset \vdash \lambda y : \alpha \to \beta . \lambda z : \alpha . yz : (\alpha \to \beta) \to \alpha \to \beta}\ \text{abs}$$

Se existisse algum problema no caso de encontrar variáveis com tipagem incongruente nas últimas premissas ou não ter mais nenhum passo, então o termo não seria bem-tipado.

3.1.6 Checagem de tipos em $\lambda_{\to}$

Seja o juizo

$$x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta$$

é necessário construir uma árvore de inferências que demonstre que $\gamma \to \beta$ é o tipo correto do termo do lado direito.

$$\dfrac{\dfrac{?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash^{?} (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta}\ ?}{}$$

Usando a regra da aplicação, tem-se:

$$\dfrac{\dfrac{?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash \lambda z : \beta . \lambda u : \gamma . z : \ ?}\ ? \qquad \dfrac{?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash yx : \ ?}\ ?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash^{?} (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta}\ \text{appl}$$

O lado direto se segue da regra da aplicação:

$$\dfrac{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash x : \alpha \to \alpha \qquad x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash y : (\alpha \to \alpha) \to \beta}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash yx : \ ?}\ \text{appl}$$

Usando essa subárvore, pode-se ver que yx possui o tipo $yx : \beta$.
O lado esquerdo se segue da abstração:

$$\dfrac{\dfrac{\dfrac{?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta \vdash \lambda u : \gamma . z : \ ?}\ ?}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash \lambda z : \beta . \lambda u : \gamma . z : \ ?}\ \text{abst}}{}$$

abstraindo novamente:

$$\dfrac{\dfrac{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta, u : \gamma \vdash z : \beta}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta \vdash \lambda u : \gamma . z : \ ?}\ \text{abst}}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash \lambda z : \beta . \lambda u : \gamma . z : \ ?}\ \text{abst}$$

Agora, é possível "descer" novamente "coletando" os tipos que foram deixados para trás:

$$\dfrac{\dfrac{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta, u : \gamma \vdash z : \beta}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta \vdash \lambda u : \gamma . z : \gamma \to \beta}\ \text{abst}}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash \lambda z : \beta . \lambda u : \gamma . z : \ ?}\ \text{abst}$$

e

$$\dfrac{\dfrac{\dfrac{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta, u : \gamma \vdash z : \beta}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta, z : \beta \vdash \lambda u : \gamma . z : \gamma \to \beta}\ \text{abst}}{x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta \vdash \lambda z : \beta . \lambda u : \gamma . z : \beta \to \gamma \to \beta}\ \text{abst}$$

Seja $\Gamma \equiv x : \alpha \to \alpha, y : (\alpha \to \alpha) \to \beta$, a árvore completa fica:

$$\dfrac{\dfrac{\dfrac{\Gamma, z : \beta, u : \gamma \vdash z : \beta}{\Gamma, z : \beta \vdash \lambda u : \gamma . z : \gamma \to \beta}\ \text{abst}}{\Gamma \vdash \lambda z : \beta . \lambda u : \gamma . z : \beta \to \gamma \to \beta}\ \text{abst} \qquad \dfrac{\Gamma \vdash x : \alpha \to \alpha \qquad \Gamma \vdash y : (\alpha \to \alpha) \to \beta}{\Gamma \vdash yx : \beta}\ \text{appl}}{\Gamma \vdash (\lambda z : \beta . \lambda u : \gamma . z)(yx) : \gamma \to \beta}\ \text{appl}$$

Dessa forma, é possível perceber que sim, a aplicação de $\lambda z : \beta . \lambda u : \gamma . z : \beta \to \gamma \to \beta$ com $yx : \beta$ possui o tipo $\gamma \to \beta$.

3.1.7 Encontrar termos em $\lambda_{\to}$

Seja o tipo $A \to B \to A$. A pergunta que fica é: é possível encontrar um termo para esse tipo? Essa pergunta é, vista do ponto da lógica, a mesma coisa que "é possível computar uma prova para essa proposição?" (Isso será visto mais adiante). Isso é a mesma coisa que: $? : A \to B \to A$. Pelas regras de inferência:

$$\dfrac{?}{? \vdash ? : A \to B \to A}\ ?$$

Supondo um termo $x : A$, pode-se escrever a árvore como:

$$\dfrac{\dfrac{?}{x : A \vdash ? : B \to A}\ ?}{x : A \vdash ? : A \to B \to A}\ \text{abst}$$

E supondo um outro termo $y : B$, pode-se escrever como:

$$\dfrac{\dfrac{\dfrac{?}{x : A, y : B \vdash ? : A}\ ?}{x : A, y : B \vdash ? : B \to A}\ \text{abst}}{x : A, y : B \vdash ? : A \to B \to A}\ \text{abst}$$

Como já existe um termo de tipo A, pode-se substituir o termo desconhecido por x:

$$\dfrac{\dfrac{x : A, y : B \vdash x : A}{x : A, y : B \vdash\ ?\ : B \to A}\ \text{abst}}{x : A, y : B \vdash\ ?\ : A \to B \to A}\ \text{abst}$$

Usando a regra da abstração:

$$\dfrac{\dfrac{x : A, y : B \vdash x : A}{x : A, y : B \vdash \lambda y.x : B \to A}\ \text{abst}}{x : A, y : B \vdash\ ?\ : A \to B \to A}\ \text{abst}$$

Novamente:

$$\dfrac{\dfrac{\dfrac{x : A, y : B \vdash x : A}{x : A, y : B \vdash \lambda y.x : B \to A}\ \text{abst}}{x : A, y : B \vdash \lambda x y.x : A \to B \to A}\ \text{abst}}$$

3.1.8 Propriedades gerais do STλC

Ficaram faltando nas definições anteriores a explicação de algumas propriedades gerais da sintaxe do STλC.

Algumas propriedades sobre os contextos:

Definição 3.6 ((Domínio, subcontexto, permutação, projeção))**.**

1. Se $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$, então o *domínio* de Γ ou $\mathtt{dom}(\Gamma)$ é a lista $(x_1, \ldots, x_n)$.
2. Um contexto Γ' é um *subcontexto* do contexto Γ, ou $\Gamma' \subseteq \Gamma$ se todas as declarações que ocorrem em Γ' também ocorrem em Γ na mesma ordem.
3. Um contexto Γ' é uma *permutação* do contexto Γ, ou $\Gamma' \subseteq \Gamma$ se todas as declarações que ocorrem em Γ' também ocorrem em Γ e vice-versa
4. Se Γ é um contexto e Φ o conjunto de variáveis, então a *projeção* de Γ em Φ, ou $\Gamma \restriction \Phi$, é o subcontexto Γ' de Γ com $\mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma) \cap \Phi$

Em uma lista, a ordem dos elementos importa.

Exemplo: Seja $\Gamma \equiv y : \sigma, x_1 : \rho_1, x_2 : \rho_2, z : \tau, x_3 : \rho_3$, então:

1. $\mathtt{dom}(\emptyset) = ()$, onde $\emptyset$ é chamado de lista vazia;
2. $\mathtt{dom}(\Gamma) = (y, x_1, x_2, z, x_3)$
3. $\emptyset \subseteq (x_1 : \rho_1, z : \tau) \subseteq \Gamma$
4. $\Gamma \restriction \{z, u, x_1\} = x_1 : \rho_1, z : \tau$

Uma propriedade importante de $\lambda_{\to}$ é a seguinte:

Lema 3.1. (Lemma das variáveis livres)
Se $\Gamma \vdash L : \sigma$, então $FV(L) \subseteq \mathtt{dom}(\Gamma)$.

Como consequência desse lemma, seja x uma variável livre que ocorre em L, então x possui um tipo, o qual é declarado no contexto Γ. Em um juizo, não é possível ocorrer confusão sobre o tipo de qualquer variável, pois todas as variáveis ligadas possuem seu tipo, antes da ligação λ.

Para provar esse lemma, é necessário usar uma técnica de prova chamada de *indução estrutural*. Essa indução ocorre da seguinte forma:

Seja $\mathcal{P}$ a propriedade geral que se quer provar para uma expressão arbitrária $\mathcal{E}$, procede-se da seguinte forma:

- Assumindo que $\mathcal{P}$ é verdadeira para toda expressão $\mathcal{E}'$ usada no construto $\mathcal{E}$ (*Hipótese Indutiva*),
- e provando que $\mathcal{P}$ também é verdadeira para $\mathcal{E}$.

Prova do Lemma: Seja $\mathcal{J} \equiv \Gamma \vdash L : \sigma$, e suponha que $\mathcal{J}$ é a conclusão final de uma derivação e assuma que o conteudo do Lemma vale para as premissas usadas para inferir a conclusão.

Pela definição das regras de inferência, existem três possibilidades de regra para conclusão: (*var*), (*appl*) e (*abst*). Provando por casos:

1. Se $\mathcal{J}$ é a conclusão da regra (*var*)
   Então $\mathcal{J}$ possui a forma $\Gamma \vdash x : \sigma$ se seguindo de $x : \sigma \in \Gamma$. O L do lemma é o x e precisamos provar que $FV(x) \subseteq dom(\Gamma)$. Mas isso é consequência direta de $x : \sigma \in \Gamma$.

2. Se $\mathcal{J}$ é a conclusão da regra (*appl*)
   Então $\mathcal{J}$ deve ter a forma $\Gamma \vdash MN : \tau$ e precisa-se provar que $FV(MN) \in dom(\Gamma)$. Por indução, a regra já é válida para as premissas de (*appl*), que são $\Gamma \vdash M : \sigma \to \tau$ e $\Gamma \vdash N : \sigma$.
   Assim, pode-se assumir que $FV(M) \subseteq dom(\Gamma)$ e $FV(N) \subseteq dom(\Gamma)$. Como $FV(MN) = FV(M) \cup FV(N)$, então $FV(MN) \subseteq dom(\Gamma)$.

3. Se $\mathcal{J}$ é a conclusão da regra (*abst*)
   Então $\mathcal{J}$ deve ter a forma $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$ e precisa-se provar que $FV(\lambda x : \sigma.M) \in dom(\Gamma)$. Por indução, a regra já é válida para a premissa de (*abst*), que é $\Gamma, x : \sigma \vdash M : \tau$.
   Assim, pode-se assumir que $FV(M) \subseteq dom(\Gamma) \cup \{x\}$. Como $FV(\lambda x : \sigma.M) = FV(M) \backslash \{x\}$, então $FV(M) \backslash \{x\} \subseteq dom(\Gamma)$.

Outras propriedades também podem ser provadas no mesmo estilo de indução:

Lema 3.2. (Afinamento, Condensação, Permutação)

1. (*Afinamento*) Sejam Γ' e Γ'' contextos tais que $\Gamma' \subseteq \Gamma''$. Se $\Gamma' \vdash M : \sigma$, então $\Gamma'' \vdash M : \sigma$
2. (*Condensação*) Se $\Gamma \vdash M : \sigma$, então também $\Gamma \restriction FV(M) \vdash M : \sigma$
3. (*Permutação*) Se $\Gamma \vdash M : \sigma$ e Γ' é uma permutação de Γ, então Γ' também é um contexto e $\Gamma' \vdash M : \sigma$.

explicação:

- O "afinamento" de um contexto é uma extensão do contexto obtida ao adicionar declarações extras com novas variáveis. O lema anterior diz que: se M é tem tipo σ em um contexto Γ', então M também terá um tipo σ em um contexto "mais fino" Γ'. Ou seja, a validade do tipo de M não muda ao adicionar novas declarações ao contexto.
- O lema da "condensação" diz que declarações $x : \rho$ podem ser retiradas de Γ caso x não ocorra livre em M. Ou seja, ele só deixa declarações relevantes à M.
- O lema da "permutação" diz que não importa o jeito que o contexto foi ordenado e também que declarações no contexto são mutualmente independentes, então não existe impedimento teórico para a permutação do contexto. (Isso não vai ser verdadeiro em todas as teorias)

Prova do (1): A prova será feita por indução no juizo $\mathcal{J} \equiv \Gamma' \vdash M : \sigma$, assumindo que $\Gamma' \subseteq \Gamma''$. Existem três casos para considerar correspondentes a cada regra de inferência:

1. Se $\mathcal{J}$ é a conclusão da regra (*var*)
   Então $\mathcal{J}$ possui a forma $\Gamma' \vdash x : \sigma$ se seguindo de $x : \sigma \in \Gamma'$. Mas se $\Gamma' \subseteq \Gamma''$, então $x : \sigma \in \Gamma''$. Desse modo, usando (*var*) tem-se que $\Gamma'' \vdash x : \sigma$.

2. Se $\mathcal{J}$ é a conclusão da regra (*appl*)
   Então $\mathcal{J}$ possui a forma $\Gamma' \vdash MN : \tau$ e precisa-se provar que $\Gamma'' \vdash MN : \tau$. Por indução, o afinamento é válido em $\Gamma' \vdash M : \sigma \to \tau$ e $\Gamma' \vdash N : \tau$. Mas, sendo assim, tem-se que $M \in \Gamma'$ e $N \in \Gamma'$, logo: $M \in \Gamma''$ e $N \in \Gamma''$ e, usando a regra (*appl*) em cima de $\Gamma'' \vdash M : \sigma \to \tau$ e $\Gamma'' \vdash N : \tau$, tem-se que $\Gamma'' \vdash MN : \tau$.

3. Se $\mathcal{J}$ é a conclusão da regra (*abst*)
   Então $\mathcal{J}$ tem que ter a forma $\Gamma' \vdash \lambda x : \rho.L : \rho \to \tau$. Temos que provar que $\Gamma' \vdash \lambda x : \rho.L : \rho \to \tau$, assumindo que $x \notin \mathtt{dom}(\Gamma'')$. Por indução na regra, temos que o "afinamento" também é válido para $\Gamma', x : \rho \vdash L : \tau$. Mas, como $x \notin \mathtt{dom}(\Gamma'')$, então podemos criar o contexto $\Gamma'', x : \rho$. E é possível ver que $\Gamma', x : \rho \subseteq \Gamma'', x : \rho$. Dessa forma, se segue que: $\Gamma'', x : \rho \vdash L : \tau$ e, através da regra, $\Gamma'' \vdash \lambda x : \rho.L : \rho \to \tau$

As provas das outras duas partes se seguem de forma similiar e são deixadas para o leitor como exercício.

Outro lema importante é o seguinte:

Lema 3.3. (Lema da Geração)

1. Se $\Gamma \vdash x : \sigma$, então $x : \sigma \in \Gamma$
2. Se $\Gamma \vdash MN : \tau$, então existe um tipo σ tal que $\Gamma \vdash M : \sigma \to \tau$ e $\Gamma \vdash N : \sigma$
3. Se $\Gamma \vdash \lambda x : \sigma.M : \rho$, então existe um τ tal que $\Gamma, x : \sigma \vdash M : \tau$ e $\rho \equiv \sigma \to \tau$.

prova: Pela inspeção das regras de inferência de $\lambda_\to$, é possível ver que não existe outra possibilidade a não ser as listadas no lema.

Lema 3.4. (Lema do subtermo) Se M é legal, então todo subtermo de M é legal.

Então, se existem Γ_1 e σ_1 tal que $\Gamma_1 \vdash M : \sigma_1$ e se L é um subtermo de M, então existem Γ_2 e σ_2 tais que $\Gamma_2 \vdash L : \sigma_2$. Com essa descrição, é possível ver que a prova também se segue da indução nas regras.

prova: Usando a indução e supondo $\Gamma \vdash x : \sigma$ como caso base, tem-se dois casos:

- Se $M \equiv NL : \tau$, então tem-se que $\Gamma \vdash NL : \tau$, onde N e L são subtermos de M. Pelo lema da geração, existe um tipo σ tal que $\Gamma \vdash N : \sigma \to \tau$ e $\Gamma \vdash L : \sigma$. Dessa forma, N e L são legais
- Se $M \equiv \lambda x.N : \rho$, então tem-se que $\Gamma \vdash \lambda x.N : \rho$, onde N é subtermo de M. Pelo lema da geração, existe um tipo τ tal que $\Gamma, x : \sigma \vdash M : \tau$ e $\rho \equiv \sigma \to \tau$. Dessa forma M é legal e $\Gamma_2 \equiv \Gamma_1, x : \sigma$.

Uma propriedade importante da Teoria dos Tipos de Church é que cada termo possui um tipo único, que pode ser descrito no seguint lema:

Lema 3.5. (Unicidade dos tipos) Assuma que $\Gamma \vdash M : \sigma$ e $\Gamma \vdash M : \tau$, então $\sigma \equiv \tau$.

Prova: Por indução na construção de M

Teorema 3.1. (Decidabilidade) Em $\lambda_{\to}$, os seguintes problemas são decidíveis:

1. Boa-tipagem: $? \vdash \mathtt{term} : ?$
2. Checagem de tipos: contexto $\vdash^?$ termo : tipo
3. Encontrar termos: contexto $\vdash ?$: tipo

Prova: A prova pode ser encontrada em (Barendregt, 1992).

3.1.9 Redução no STλC

Até agora, não havia sido definido o comportamento da β-redução no STλC. Para fazer isso, é necessário introduzir o seguinte lema:

Lema 3.6. (Lema da Substituição) Seja $\Gamma', x : \sigma, \Gamma'' \vdash M : \tau$ e $\Gamma' \vdash N : \sigma$, então $\Gamma', \Gamma'' \vdash M[x := N] : \tau$.

Esse lema diz que se em um termo legal M for substituido todas as ocorrências da variável do contexto x por um termo N de mesmo tipo que x, então o resultado $M[x := N]$ possui o mesmo tipo que M.

prova: Usando indução em cima do juizo $\mathcal{J} \equiv \Gamma', x : \sigma, \Gamma'' \vdash M : \tau$.

1. Se $\mathcal{J}$ é a conclusão da regra (*var*)
   Então $\mathcal{J}$ possui a forma $\Gamma', x : \sigma, \Gamma'' \vdash x : \sigma$. Se o contexto é bem formado, então $x : \sigma$ não está em Γ'' e $x \notin FV(N)$. Com isso, pode-se inferir que $x[x := N] : \sigma$.

2. Se $\mathcal{J}$ é a conclusão da regra (*appl*)
   Então $\mathcal{J}$ possui a forma $\Gamma' \vdash MN : \tau$, pela regra de inferência, temos dois juizos $\mathcal{J}' \equiv \Gamma' \vdash M : \rho \to \tau$ e $\mathcal{J}'' \equiv \Gamma' x : \sigma \vdash N : \rho$ para os quais vale o lema, logo supondo $\Gamma' \vdash L : \sigma$, temos que: $\Gamma', \Gamma'' \vdash M[x := N] : \rho \to \tau$ e $\Gamma', \Gamma'' \vdash N[x := L] : \rho$. Usando a regra da aplicação, temos: $\Gamma', \Gamma'' \vdash (M[x := L])N(x := L) : \tau$ que é a mesma coisa que $\Gamma', \Gamma'' \vdash (MN)(x := L) : \tau$. □

3. Se $\mathcal{J}$ é a conclusão da regra (*abst*)
   Então $\mathcal{J}$ tem que ter a forma $\Gamma' \vdash \lambda u : \rho.L : \rho \to \tau$. Logo existe um outro juizo $\mathcal{J}' \equiv \Gamma', x : \sigma, \Gamma'', u : \rho \vdash L : \tau$. Mas em $\mathcal{J}'$, $x : \sigma$ não pode ocorrer em Γ', logo como $\Gamma' \vdash N : \sigma$, $x \notin FV(N)$. Usando o lema, temos que $\Gamma', \Gamma'', u : \rho \vdash L[x := N] : \tau$. Usando a regra da abstração: $\Gamma', \Gamma'' \vdash \lambda u : \rho.(L[x := N]) : \rho \to \tau$, que é o mesmo que $\Gamma', \Gamma'' \vdash (\lambda u : \rho.L)[x := N] : \rho \to \tau$. □

Tendo definido a substituição, pode-se definir a β-redução:

Definição 3.7. (β-redução de passo único para $\Lambda_{\mathbb{T}}$)

1. (Base) $(\lambda x : \sigma.M)N \to_\beta M[x := N]$
2. (Compatibilidade) Como na definição 1.10

Como os tipos não são importantes no processo de β-redução, o Teorema de Church-Rosser também se torna válido no $\lambda_\to$:

Teorema 3.2. (Teorema de Church-Rosser) A propriedade de Church-Rosser também é válida para $\lambda_\to$

Corolário 3.1. Suponha que $M =_\beta N$, então existe um L tal que $M \twoheadrightarrow_\beta L$ e $N \twoheadrightarrow_\beta L$

Lema 3.7. (Redução do sujeito) Se $\Gamma \vdash L : \rho$ e se $L \twoheadrightarrow_\beta L'$, então $\Gamma \vdash L' : \rho$.

Esse lema final mostra que a β-redução não afeta a tipabilidade e não muda o tipo do termo afetado, logo o mesmo contexto inicial serve para inferir.
Prova:

Teorema 3.3. (Teorema da normalização forte) Todo termo legal M é fortemente normalizável

Esse teorema garante que não existam termos que não são reduzíveis, ou seja, todo termo legal em $\lambda_\to$ possi uma forma normal e nem todo termo legal possui um ponto fixo. Isso faz com que o STλC não seja turing-completo. Essa característica não é muito desejável na implementação de linguagens de programação, pois na vida real, é necessário implementar códigos que podem não terminar. Por esse motivo, é necessário formar extensões do cálculo para que ele funcione nesses casos.

O fato do universo de funções legais possíveis ser reduzido bastante no STλC fez com que pesquisas em modelos partindo do Cálculo λ não tipado fossem desenvolvidas. Esses modelos como trabalhados na subseção 1.2 possuem vantagens (e desvantagens) em relação à tipagem.

3.2 Extensões ao STλC e as Teorias dos Tipos Simples

Partindo do cálculo λ não-tipado visto no capítulo anterior, sua tipagem natural é de fato a vista anteriormente usando somente tipos funcionais. Mas alguns comportamentos que os tipos deveriam ter acabam não sendo construidos ou sendo construidos de forma muito complicada e pouco natural. Alguns tipos adicionais são requeridos para construir uma teoria de tipos mais robusta.

A construção original da teoria dos tipos simples feita em (CHURCH, 1940) já é, por si só, mais complexa que a teoria vista aqui. Dessa forma, esse seção se preocupa em extender o STλC visto até então para de fato a Teoria dos Tipos Simples.

Nessa seção serão introduzidos um tipo novo por vez, discutindo suas motivações, regras de inferência e exemplos de derivação. No final, eles serão reunidos para discutir a teoria dos tipos simples.

3.2.1 Adendos sobre o tipo funcional

Uma pergunta possível de se fazer é se a definição 3.1 pode ser reescrita tendo em mente as regras de inferência, e a resposta é sim. Para isso, é necessária a seguinte notação: seja α um tipo simples, a notação desse fato seria $\alpha \in \mathbb{T}$, agora pode-se usar a regra de

$$\alpha \text{ Type}$$

. Logo, é possível reformular a definição 3.1 da seguinte forma:

Definição 3.8. Os tipos simples possuem as seguintes regras de derivação:

1. $\dfrac{}{\vdash \sigma \text{ Type}}$ para $\sigma \in \mathbb{V}$
2. $\dfrac{\vdash \sigma \text{ Type} \qquad \vdash \tau \text{ Type}}{\vdash \sigma \to \tau \text{ Type}}$

Outras formas de trabalhar com o conceito de tipos dessa forma serão introduzidos no próximo capítulo. Esse tipo de regra de inferência na teoria dos tipos simples serve para localizar de forma visual a formação dos tipos, por isso normalmente são chamadas de *regras de formação*.

Uma outra forma de generalizar a construção feita anteriormente do tipo funcional é pensar as regras de inferência introduzidas anteriormente, a (*abst*) e (*appl*), como regras de introdução (*intro*) e eliminação (*elim*), respectivamente. Essas regras serão colocadas aqui:

$$(\to\text{-intro}) \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x : \alpha.M : \alpha \to \beta} \qquad (\to\text{-elim}) \frac{\Gamma \vdash M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

Outras duas regras de inferência importantes para a construção feita aqui são construidas em cima da β-redução e da η-redução. Para lembrar, a β-redução de um termo $(\lambda x.M)N$ é o termo $M[N/x]$ e a η-redução do termo $\lambda x.Mx$ é o termo M. Essas duas reduções podem ser escritas equacionalmente: seja $=$ uma relação de igualdade entre termos de um mesmo tipo, então

$$\beta \frac{\Gamma, x : \alpha \vdash M : \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash (\lambda x : \alpha.M)N = M[N/x] : \beta} \qquad \eta \frac{\Gamma \vdash M : \alpha \to \beta}{\Gamma \vdash \lambda x : \alpha.Mx = M : \alpha \to \beta}$$

Nas exposições dos novos tipos nesse capítulo, os tipos serão introduzidos na ordem vista anteriormente sempre que possível: formação, introdução, eliminação, β-redução e η-redução.

3.2.2 O tipo de produto e o tipo unitário ($\lambda 1_\times$)

Uma primeira extensão do STλC é a teoria dos tipos simples possuindo os tipos funcionais, tipos de produto e o tipo unitário.

I. Tipo unitário

O tipo unitário, também chamado de tipo vazio, é um tipo postulado, ou seja sua formação se comporta mais ou menos como na definição 3.8(1) vista anteriormente:

$$\frac{}{\vdash \mathbb{1}\ \mathrm{Type}}$$

Sua regra de introdução é a seguinte:

$$\frac{}{\vdash () : \mathbb{1}}$$

O nome unitário é dado porque ele é somente habitado por um único termo denotado por $()$ ou por vezes $\langle\rangle$. Para postular que esse termo é único, é necessário ainda introduzir a seguinte regra:

$$\eta\ \frac{\Gamma \vdash M : \mathbb{1}}{\Gamma \vdash M = () : \mathbb{1}}$$

Sua regra de eliminação vai ser introduzida aqui da seguinte forma:

$$\frac{\vdash c : C}{x : \mathbb{1} \vdash \mathrm{let}() = x \text{ in } c : C}$$

Essa regra muda um pouco na teoria dos tipos dependentes mas isso será visto posteriormente.

A regra que falta é a da β-redução:

$$\beta\ \frac{\vdash () : \mathbb{1}}{\vdash (\mathrm{let}() = () \text{ in } c) = c : C}$$

II. Tipos de Produtos

Os tipos de produtos são tipos que se comportam como pares ordenados. Sua notação de tipo é denotada pelo simbolo $\times$ como nos pares ordenados normalmente, sendo sua regra de formação:

$$\frac{\vdash \sigma\ \mathrm{Type} \qquad \vdash \tau\ \mathrm{Type}}{\vdash \sigma \times \tau\ \mathrm{Type}}$$

Os elementos de um tipo $\sigma \times \tau$ são pares $\langle v, u\rangle$ formados por elementos $v : \sigma$ e $u : \tau$. Para eliminar esse tipo, é somente necessário pegar o primeiro elemento ou o segundo elemento do par e para isso são usados os operadores π_1 e π_2, respectivamente, chamados de *projeções*.

Logo, as suas regras de introdução e eleminação são:

$$(\times\text{-intro})\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N\rangle : \sigma \times \tau}$$

$$(\times\text{-elim-1})\ \frac{\Gamma \vdash P : \sigma \times \tau}{\Gamma \vdash \pi_1 P : \sigma} \qquad (\times\text{-elim-2})\ \frac{\Gamma \vdash P : \sigma \times \tau}{\Gamma \vdash \pi_2 P : \tau}$$

As suas regras de β-redução sãs as seguintes:

$$\beta\ \frac{\Gamma \vdash \langle M, N\rangle : \sigma \times \tau}{\Gamma \vdash \pi_1\langle M, N\rangle = M : \sigma} \qquad \beta\ \frac{\Gamma \vdash \langle M, N\rangle : \sigma \times \tau}{\Gamma \vdash \pi_2\langle M, N\rangle = N : \tau}$$

E a sua regra de η-redução é a seguinte:

$$\eta\ \frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \langle \pi_1 t, \pi_2 t\rangle = t : \sigma \times \tau}$$

As regras de substituição para o tipo de produtos são as seguintes:

$$(\pi_1 P)[L/v] \equiv \pi_1(P[L/v])$$
$$(\pi_2 P)[L/v] \equiv \pi_2(P[L/v])$$
$$\langle M, N\rangle[L/v] \equiv \langle M[L/v], N[L/v]\rangle$$

3. Exemplo

Um exemplo básico do tipo de produto é a construção do seguinte tipo: $\sigma \times \tau \to \tau \times \sigma$. Ele significa basicamente que a operação de produto cartesiano é reflexiva e pode ser provado da seguinte forma:

$$\frac{?}{\Gamma \vdash ? : \sigma \times \tau \to \tau \times \sigma}$$

Usando a regra (→-intro):

$$(\to\text{-intro})\ \frac{?\ \dfrac{?}{\Gamma, t : \sigma \times \tau \vdash ? : \tau \times \sigma}}{\Gamma \vdash \lambda t : \sigma \times \tau.? : \sigma \times \tau \to \tau \times \sigma}$$

Agora é necessário gerar um par onde o primeiro termo tem tipo τ e o segundo tem tipo σ.

$$\begin{array}{c}(\times\text{-intro})\ \dfrac{\Gamma, t : \sigma \times \tau \vdash \langle ?, ?\rangle : \tau \times \sigma}{\Gamma, t : \sigma \times \tau \vdash \langle ?, ?\rangle : \tau \times \sigma} \\ (\to\text{-intro})\ \dfrac{}{\Gamma \vdash \lambda t : \sigma \times \tau.\langle ?, ?\rangle : \sigma \times \tau \to \tau \times \sigma}\end{array}$$

Sabendo que o tipo de $\pi_1 t$ é σ e o tipo de $\pi_2 t$ é τ, sabe-se que $\langle \pi_2 t, \pi_1 t\rangle$ tem o tipo $\tau \times \sigma$ que é o tipo que se quer encontrar, logo:

$$\begin{array}{c}(\times\text{-intro})\ \dfrac{\Gamma, t : \sigma \times \tau \vdash \langle \pi_2 t, \pi_1 t\rangle : \tau \times \sigma}{\Gamma, t : \sigma \times \tau \vdash \langle \pi_2, \pi_1 t\rangle : \tau \times \sigma} \\ (\to\text{-intro})\ \dfrac{}{\Gamma \vdash \lambda t : \sigma \times \tau.\langle \pi_2, \pi_1 t\rangle : \sigma \times \tau \to \tau \times \sigma}\end{array}$$

Uma prova para o leitor é mostrar que $(\alpha \times \beta) \times \gamma \to \alpha \times (\beta \times \gamma)$ é habitado.

Outro fato interessante que pode ser provado é o *currying* que pode ser resumido no seguinte tipo: $((\alpha \times \beta) \to \gamma) \to (\alpha \to \beta \to \gamma)$. Isso mostra que caso a pessoa tenha uma função que pega um par e retorna um tipo, isso é o mesmo que uma função que pega cada elemento do par em um momento e retorna esse tipo.

A sua construção é feita da seguinte forma:

$\lambda f : ((\alpha \times \beta) \to \gamma).\lambda x : \alpha.\lambda y : \beta.f\langle x, y\rangle$

3.2.3 O tipo de produto disjunto e tipo vazio ($\lambda 1_{(\times,+)}$)

I. O Tipo Vazio

Enquanto o tipo unitário é caracterizado pela característica de só possuir um único termo, enquanto o tipo vazio é caracterizado de forma análoga por ser um tipo que não possui nenhum termo.

A sua regra de formação é a seguinte:

$$\frac{}{\vdash \emptyset \text{ Type}}$$

O tipo vazio, como seu próprio nome indica, não possui uma regra de introdução de termos.

II. O Tipo produto disjunto

O tipo produto disjunto é uma formação na teoria dos tipos análoga à formação do produto disjunto na teoria dos conjuntos. Sua regra de formação é a seguinte:

$$\frac{\vdash \sigma \text{ Type} \qquad \vdash \tau \text{ Type}}{\vdash \sigma + \tau \text{ Type}}$$

Sua regra de introdução, diferente do tipo de produto falado anteriormente, não depende da presença de dois termos, um para cada tipo no produto, mas somente de um dos dois termos:

$$(+\text{-intro-1}) \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \kappa M : \sigma + \tau} \; (+\text{-intro-2}) \frac{\Gamma \vdash N : \tau}{\Gamma \vdash \kappa' N : \sigma + \tau}$$

Normalmente o operador κ é chamado de $\mathtt{inl}$ enquanto o operador κ' é chamado de $\mathtt{inr}$ na teoria dos tipos dependentes, mas aqui está sendo usada a notação de (JACOBS, 1999). κ é denominado de *coprojeção*

Sua regra de eliminação é a mais complexa introduzida aqui, sendo a seguinte:

$$(+\text{-elim}) \frac{\Gamma \vdash P : \sigma + \tau \qquad \Gamma, x : \sigma \vdash Q : \rho \qquad \Gamma, y : \tau \vdash R : \rho}{\Gamma \vdash \text{unpack } P \text{ as } [\kappa x \text{ in } Q, \kappa' y \text{ in } R] : \rho}$$

A interpretação dessa regra é a seguinte: olhe para P, se P está em σ então faça Q com P sendo x, caso P esteja em τ, faça R com P como y.

Para o unpack, as regras de β-redução se seguem da interpretação da eliminação:

$$\beta \frac{\Gamma \vdash M : \sigma \qquad \Gamma, x : \sigma \vdash Q : \rho \qquad \Gamma, y : \tau \vdash R : \rho}{\Gamma \vdash \text{unpack } \kappa M \text{ as } [\kappa x \text{ in } Q, \kappa' y \text{ in } R] = Q[M/x] : \rho}$$

$$\beta \frac{\Gamma \vdash N : \tau \qquad \Gamma, x : \sigma \vdash Q : \rho \qquad \Gamma, y : \tau \vdash R : \rho}{\Gamma \vdash \text{unpack } \kappa' N \text{ as } [\kappa x \text{ in } Q, \kappa' y \text{ in } R] = R[N/y] : \rho}$$

A regra de η redução é a seguinte:

$$\eta \frac{\Gamma \vdash P : \sigma + \tau \qquad \Gamma, z : \sigma + \tau \vdash R : \rho}{\Gamma \vdash \text{unpack } P \text{ as } [\kappa x \text{ in } R[(\kappa x)/z], \kappa' y \text{ in } R[(\kappa' y)/z]] = R[P/z] : \rho}$$

O operador unpack também pode ser chamado de *match*, por ter essa noção de casar as coprojeções de P para dois casos diferentes.

III. Exemplo

Uma prova interessante é a que os coprodutos são distributivos em relação à produtos, ou seja $((\sigma \times \tau) + (\sigma \times \rho)) \to \sigma \times (\tau + \rho)$

Para isso, é necessário construir a seguinte árvore de prova:

$$? \ \frac{?}{\Gamma \vdash ? : ((\sigma \times \tau) + (\sigma \times \rho)) \to \sigma \times (\tau + \rho)}$$

Usando a abstração:

$$(\to\text{-intro}) \ \frac{? \ \dfrac{?}{\Gamma, u : (\sigma \times \tau) + (\sigma \times \rho) \vdash ? : \sigma \times (\tau + \rho)}}{\Gamma \vdash \lambda u : (\sigma \times \tau) + (\sigma \times \rho).? : ((\sigma \times \tau) + (\sigma \times \rho)) \to \sigma \times (\tau + \rho)}$$

Ao "abrir" u, tem-se duas possibilidades:

- u é do tipo $\sigma \times \tau$, ou seja $\pi_1 u = \sigma$ e $\pi_2 u = \tau$, com $pi_1 u$ tem-se a primeira parte do tipo objetivo, já $\tau + \rho$ é possível ser gerado por $\kappa(\pi_2 u)$
- u é do tipo $\sigma \times \rho$, ou seja $\pi_1 u = \sigma$ e $\pi_2 u = \rho$, da mesma forma tem-se a primeira parte do tipo objetivo de cara, já $\tau + \rho$ é gerado por $\kappa'(\pi_2 u)$.

Logo o termo final é o seguinte:

$$\text{unpack } u \text{ as } [\kappa x \text{ in } \langle \pi_1 x, \kappa(\pi_2 x)\rangle, \kappa' y \text{ in } \langle \pi_1 y, \kappa'(\pi_2 y)\rangle] : \sigma \times (\tau + \rho)$$

Logo a árvore final é:

$$(\to\text{-intro}) \ \frac{(+\text{-elim}) \ \dfrac{\Gamma \vdash u : (\sigma \times \tau) + (\sigma \times \rho) \qquad \Gamma, x : (\sigma \times \tau) \vdash : \langle \pi_1 x, \kappa(\pi_2 x)\rangle : \sigma \times (\tau + \rho) \qquad \Gamma, y : (\sigma \times \rho) \vdash \langle}{\Gamma, u : (\sigma \times \tau) + (\sigma \times \rho) \vdash \text{unpack } u \text{ as } [\kappa x \text{ in } \langle \pi_1 x, \kappa(\pi_2 x)\rangle, \kappa' y \text{ in } \langle \pi_1 y, \kappa'(\pi_2 y)\rangle}}{\Gamma \vdash \lambda u : (\sigma \times \tau) + (\sigma \times \rho).\text{unpack } u \text{ as } [\kappa x \text{ in } \langle \pi_1 x, \kappa(\pi_2 x)\rangle, \kappa' y \text{ in } \langle \pi_1 y, \kappa'(\pi_2 y)\rangle] : \sigma \times (\tau + \rho) : (}$$

O resto da dedução da árvore fica como exercício para o leitor.

3.2.4 O Tipo dos Números Naturais

Foi visto na parte do cálculo λ não tipado a codificação dos números naturais utilizando o método da codificação de Church. Esse tipo de codificação é útil caso se queira trabalhar com uma versão reduzida da teoria dos tipos e possui propriedades interessantes que serão exploradas na parte do Sistema F no próximo capítulo. Porém, uma de suas desvantagens é que muitas vezes os cálculos podem demandar muito, principalmente se essa codificação for utilizada em linguagens de programação. Dessa forma, um jeito mais fácil de desenvolver os números naturais é postulando-os da mesma forma que foi feito anteriormente.

Primeiro, é necessário postular a existência de um tipo dos números naturais:

$$\frac{}{\vdash \mathbb{N}\ \text{Type}}$$

A sua regra de introdução segue os axiomas de Peano sobre o número 0 e a função sucessor $\mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ que leva um número n ao seu sucessor $n + 1$. Na forma de regras de inferência:

$$\frac{}{\vdash 0 : \mathbb{N}} \qquad \frac{}{\vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

Essa forma de definir a função sucessor é encontrada em por exemplo (RIJKE, 2022). Uma outra forma de definir o sucessor é a vista em (**??**) como:

$$\frac{\vdash n : \mathbb{N}}{\vdash S(n) : \mathbb{N}}$$

A regra de eliminação do tipo $\mathbb{N}$ é baseado no método da indução que diz o seguinte: Seja ϕ um predicado tal que

(1) $\phi(0)$ é verdade, e

(2) para todo número natural n, se $\phi(n)$ é verdadeiro, então $\phi(S(n))$ também o é

Logo $\phi(n)$ é verdadeiro para todo número natural n.

Para a teoria dos tipos simples, o predicado ϕ se torna um tipo P e sua regra de eliminação se torna:

$$(\mathbb{N}\text{-elim})\ \frac{\Gamma, n : \mathbb{N} \vdash P\ \text{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_S : \mathbb{N} \to P \to P}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S) : \mathbb{N} \to P}$$

A interpretação dessa regra é a seguinte:

- Premissa 1: Dado que de um termo $n : \mathbb{N}$ é possível derivar o tipo P
- Premissa 2: Dado que P é válido para 0, tendo como prova p_0
- Premissa 3: equivale à regra (2) para o predicado ϕ anterior
- Conclusão: o termo $\mathrm{ind}_{\mathbb{N}}(p_0, p_S)$ é a função que pega as duas provas da indução e gera o tipo P.

Como sua regra de eliminação está relacionada ao conceito de indução, o tipo $\mathbb{N}$ é chamado de um *tipo indutivo*. Não só $\mathbb{N}$ é um tipo, indutivo, mas também outros tipos como o tipo unitário e o tipo dos booleanos, que será visto posteriormente, podem ser vistos como tipos indutivos.

As regras de computação, para β-redução, são escritas uma para o caso base, p_0, e outra para o passo indutivo, p_S:

$$\beta\ \frac{\Gamma, n : \mathbb{N} \vdash P\ \text{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_S : \mathbb{N} \to P \to P}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S, 0_{\mathbb{N}}) = p_0 : P}$$

$$\beta\ \frac{\Gamma, n : \mathbb{N} \vdash P\ \text{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_S : \mathbb{N} \to P \to P}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S, \mathrm{succ}_{\mathbb{N}}(n)) = p_S(n, \mathrm{ind}_{\mathbb{N}}(p_0, p_S, n)) : P}$$

Exemplo:

Definição 3.9 (Adição, (RIJKE, 2022))**.** Definindo uma função

$$add_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

que satisfaça a especificação:

$$add_{\mathbb{N}}(m, 0_{\mathbb{N}}) = m$$

$$add_{\mathbb{N}}(m, succ_{\mathbb{N}}(n)) = succ_{\mathbb{N}}(add_{\mathbb{N}}(m, n))$$

$add_{\mathbb{N}}(m, n)$ será denotado por $m + n$

Para sua construção, é necessário usar a regra de eliminação para $P = \mathbb{N}$. Logo, as premissas se tornam:

$$m : \mathbb{N} \vdash \mathbb{N}$$

$$m : \mathbb{N} \vdash add - zero_{\mathbb{N}}(m) : \mathbb{N}$$

$$m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

e sua conclusão é:

$$m : \mathbb{N} \vdash add_{\mathbb{N}}(m) := ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}$$

O termo $add - zero_{\mathbb{N}}(m)$ é o próprio m, já para a adição do sucessor, é necessário saber qual o comportamento da soma para o sucesso, que é a seguinte:

$$add_{\mathbb{N}}(m, succ_{\mathbb{N}}(n)) = succ_{\mathbb{N}}(add_{\mathbb{N}}(m, n))$$

Logo, $add - succ_{\mathbb{N}}(m)$ é o mesmo que o sucessor da adição:

$$add - succ_{\mathbb{N}}(m) := \lambda n.succ_{\mathbb{N}}$$

A árvore de derivação desse termo é o seeguinte:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathbb{N} \to \mathbb{N} \qquad \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}{n : \mathbb{N} \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N}, n : \mathbb{N} \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N} \vdash \lambda n.succ_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}}{m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) := \lambda n.succ_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}$$

A regra de inferência final se torna:

$$\dfrac{\dfrac{\dfrac{\vdots}{m : \mathbb{N} \vdash \mathbb{N}\ \text{Type}} \quad \dfrac{\vdots}{m : \mathbb{N} \vdash add - zero_{\mathbb{N}}(m) := m : \mathbb{N}} \quad \dfrac{\vdots}{m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}}{m : \mathbb{N} \vdash ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N} \vdash add_{\mathbb{N}}(m) := ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}$$

3.2.5 O Tipo dos Booleanos

O tipo booleano é um tipo que possui dois elementos: um que corresponde ao valor verdadeiro na lógica proposicional e outro que corresponde ao valor falso.

$$\frac{}{\vdash \mathtt{Bool}\ \mathtt{Type}}$$

Sua regra de introdução é:

$$(\mathtt{bool\text{-}intro})\ \frac{}{\vdash \mathtt{false} : \mathtt{Bool}} \qquad (\mathtt{bool\text{-}intro})\ \frac{}{\vdash \mathtt{true} : \mathtt{Bool}}$$

O tipo dos booleanos é um tipo indutivo, logo sua eliminação segue uma forma parecida à regra dos naturais:

$$(\mathtt{Bool\text{-}elim})\ \frac{\Gamma, b : \mathtt{Bool} \vdash P\ \mathtt{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_1 : P}{\Gamma \vdash \mathtt{ind}_{\mathtt{Bool}}(p_0, p_1) : \mathtt{Bool} \to P}$$

Suas regras de β-redução são as seguintes:

$$\beta\ \frac{\Gamma, b : \mathtt{Bool} \vdash P\ \mathtt{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_S : \mathtt{Bool} \to P \to P}{\Gamma \vdash \mathtt{ind}_{\mathtt{Bool}}(p_0, p_1, \mathtt{false}) = p_0 : P}$$

$$\beta\ \frac{\Gamma, b : \mathtt{Bool} \vdash P\ \mathtt{Type} \qquad \Gamma \vdash p_0 : P \qquad \Gamma \vdash p_S : \mathtt{Bool} \to P \to P}{\Gamma \vdash \mathtt{ind}_{\mathtt{Bool}}(p_0, p_1, \mathtt{true}) = p_1 : P}$$

Exemplo

Um exemplo é a função de negação, que pega $\mathtt{true}$ e retorna $\mathtt{false}$, e vice-versa. Essa função pode ser construida usando a regra da eliminação dos booleanos usando $P := \mathtt{Bool}$:

$$(\mathtt{Bool\text{-}elim})\ \frac{b : \mathtt{Bool} \vdash \mathtt{Bool}\ \mathtt{Type} \qquad b : \mathtt{Bool} \vdash \mathtt{false} : \mathtt{Bool} \qquad b : \mathtt{Bool} \vdash \mathtt{true} : \mathtt{Bool}}{\vdash \mathtt{ind}_{\mathtt{Bool}}(\mathtt{true}, \mathtt{false}) : \mathtt{Bool} \to \mathtt{Bool}}$$

Pelas regras de β-redução é possível ver que essa construção está correta.

3.2.6 O Sistema T de Gödel

O *Sistema T de Gödel* é uma teoria dos tipos baseada na chamada *Interpretação da Dialectica* realizada por Gödel no seu artigo "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes" (Sobre uma extensão do ponto de vista finitário ainda não utilizada) publicado na revista Dialectica em 1958 (GÖDEL, 1980).

Nesse artigo, Gödel procura uma nova forma de provar a consistência da teoria dos números, usando como objetos primários construções abstratas como provas, proposições, etc. O sistema desenvolvido por Gödel utiliza uma construção de tipos próxima à construção feita por Alonzo Church, mas com algumas diferenças. Gödel não cita o cálculo λ em seu artigo mas cita a teoria dos números intuicionista de Heyting.

A primeira interpretação no cálculo λ do sistema T de gödel foi realizada no artigo de W.W.Tait "Intensional Interpretations of Functionals of Finite type" (Interpretações intensionais de funcionais de tipo finito) de 1967 (TAIT, 1967).

Uma formulação mais contemporânea do Sistema T de Gödel é feita em (GIRARD; TAYLOR; LAFONT, 1989) e será a usada aqui.

Definição 3.10 (Tipos do Sistema T de Gödel, (GIRARD; TAYLOR; LAFONT, 1989))**.**

1. Tipos atômicos $T_1, T_2, \ldots, T_n$ são tipos
2. Se U e V são tipos, então $U \times V$ e $U \to V$ são tipos
3. O tipo $\mathtt{Int}$ é o tipo dos números inteiros
4. O tipo $\mathtt{Bool}$ é o tipo dos booleanos

Já os termos são definidos da seguinte forma:

Definição 3.11 (Termos do Sistema T, (GIRARD; TAYLOR; LAFONT, 1989))**.**

1. As variáveis $x_0^T, \ldots, x_n^T, \ldots$ são termos de tipo T
2. Se u e v são termos de tipos U e V, respectivamente, $\langle u, v \rangle$ é um termo de tipo $U \times V$
3. Se t é um termo de tipo $U \times V$, então $\mathtt{pi}^1 t$ e $\pi^2 t$ são termos de tipo U e V, respectivamente
4. Se v é um termo de tipo V e x_n^U é uma variável de tipo U, então $\lambda x_n^U.v$ é um termo de tipo $U \to V$
5. Se t e u são termos de tipos $U \to V$ e U respectivamente, então tu é um termo do tipo V
6. ($\mathtt{Int}$-introdução)
   - O é uma constante de tipo $\mathtt{Int}$
   - se t tem tipo $\mathtt{Int}$, St tem tipo $\mathtt{Int}$
7. ($\mathtt{Int}$-eliminação) se u, v e t possuem tipo U, $U \to (\mathtt{Int} \to U)$ e $\mathtt{Int}$, então $Ruvt$ tem tipo U
8. ($\mathtt{Bool}$-introdução) T e F são constantes de tipo $\mathtt{Bool}$
9. ($\mathtt{Bool}$-eliminação) se u, v e t possuem tipo U, U e $\mathtt{Bool}$, então $Duvt$ tem tipo U

As duas definições anteriores preservam a notação de Girard, mas salvo excessões essa notação não será mantida, por quebrar a continuidade da notação feita até então.

O leitor que viu as extensões até então vai perceber que o Sistema T de Gödel é uma extensão da STλC que possui os tipos $\times, \mathbb{N}$ e $\mathtt{Bool}$. A diferença é a nomeclatura ($\mathtt{Int}$ no lugar de $\mathbb{N}$) e as regras de eliminação para os tipos indutivos, pois, pela interpretação de (GIRARD; TAYLOR; LAFONT, 1989), R é o operador de recursão e D é interpretado como "se ... então ... senão".

Definição 3.12 (Regras de Conversão, (GIRARD; TAYLOR; LAFONT, 1989))**.**

1. $RuvO \to_\beta u$
2. $Ruv(St) \to_\beta v(Ruvt)t$

3. $\mathsf{D}u\nu\mathsf{T} \rightarrow_\beta u$

4. $\mathsf{D}u\nu\mathsf{F} \rightarrow_\beta \nu$

Exemplos de booleanos:

1. $neg(u) = \mathsf{DFT}u$

2. $disj(u, \nu) = \mathsf{DT}\nu u$

3. $conj(u, \nu) = \mathsf{D}\nu\mathsf{F}u$

Exemplos de inteiros:

1. $add(x, y) = \mathsf{R}x(\lambda z^{Int}.\lambda z'^{Int}.\mathsf{S}z)y$

Teorema 3.4 (Teorema da Normalização Forte, (GIRARD; TAYLOR; LAFONT, 1989))**.** No sistema T, todas as sequências de redução são finitas e resultam na mesma forma normal

Prova: ver (GIRARD; TAYLOR; LAFONT, 1989) seção 7.2

3.2.7 A teoria simples de tipos de Church

A teoria simples de tipos é a primeira formulação da teoria dos tipos simples em cima do cálculo λ não tipado, realizado no artigo "A formulation of the simple theory of types" (Uma formulação da teoria simples de tipos) (CHURCH, 1940). Nessa teoria, Church construi um sistema axiomático de tipos em cima do cálculo λ. Nessa seção será feito um esforço de exposição dessa teoria como em seu primeiro artigo, apontando as suas diferenças em relação às formulações atuais da teoria dos tipos simples.

Para começar, a teoria simples de tipos de Church é um sistema axiomático, logo ele não faz uso de regras de inferência como foi feito anteriormente.

As seguintes seções seguem a ordem do artigo original

I. Hierarquia de Tipos

Nessa primeira parte, Church desenvolve a sintaxe dos *simbolos de tipos*:

> "A classe de *símbolos de tipos* é descrita pelas regras que ι e o são cada um simbolos de tipos e que se α e β são simbolos de tipos, então $(\alpha\beta)$ são simbolos de tipos." (CHURCH, 1940, p. 56)

Letras gregas são usadas para variáveis de tipos (a exceção de ι, o e λ) e os tipos $\alpha\beta$ são associativos a esquerda.

A interpretação dos tipos é a seguinte:

> "Na interpretação da teoria, subíndices devem indicar que o tipo da variável é constante, o sendo o tipo das proposições, ι o tipo dos individuos e $(\alpha\beta)$ sendo o tipo de funções com uma variável para os quais a faixa da variável independente compreende o tipo β e a faixa da variável dependente é contida no tipo α" (CHURCH, 1940, p. 57)

Logo, é possível ver que $\alpha\beta$ corresponde ao tipo $\beta \to \alpha$ definido nas seções anteriores. o não corresponde exatamente ao tipo dos booleanos, sendo normalmente tratado como o tipo PROP, enquanto ι representa variáveis individuais específicas.

Exemplo: O tipo $o\iota\iota$ é o mesmo que $\iota \to \iota \to o$.

II. Formulas bem formadas

Uma vez definida a construção dos tipos, Church define a linguagem, da seguinte forma:

> "Os *simbolos primitivos* são dados pela seguinte lista infinita:
>
> $$\lambda, (,), N_{oo}, A_{ooo}, \Pi_{o(o\alpha)}, \iota_{\alpha(o\alpha)}, a_\alpha, b_\alpha, \ldots, z_\alpha, \bar{a}_\alpha, \bar{b}_\alpha, \ldots$$
>
> " (CHURCH, 1940, p. 57)

$N_{oo}, A_{ooo}, \Pi_{o(o\alpha)}$, e $\iota_{\alpha(o\alpha)}$ (Atualmente, $N : o \to o, A : o \to o \to o, \Pi : (\alpha \to o) \to o$ e $\iota : (\alpha \to o) \to \alpha$) são chamados de *constantes* e o resto de *variáveis*.

A definição de formulas bem formadas é a seguinte:

> "(1) uma formula consistindo de um símbolo único próprio é uma formula bem-formada e possui tipo indicado pelo subíndice; (2) se x_β é uma variável com subíndice β e M_α é uma formula bem-formada com tipo α, então $(\lambda x_\beta M_\alpha)$ é uma formula bem formada de tipo $\alpha\beta$; (3) se $F_{\alpha\beta}$ e A_β são formulas bem-formadas de tipos $\alpha\beta$ e β, respectivamente, então $(F_{\alpha\beta} A_\beta)$ é uma formula bem-formada de tipo α" (CHURCH, 1940, p. 57)

Essa definição constroi o início da β-redução, que será definida na próxima parte.

Church usa a notação $\to$ no artigo da mesma forma que hoje em dia se usa $\equiv$ ou $:=$, como uma notação para "é uma abreviação de" e através disso ele constroi os operadores da lógica, a codificação de Church para os números naturais e os combinadores

Os operadores da lógica são os seguintes:

$$
\begin{aligned}
&[\sim A_o] \to N_{oo} A_o \\
&[A_o \vee B_o] \to A_{ooo} A_o B_o \\
&[A_o B_0] \to [\sim [[\sim A_o] \vee [\sim B_o]]] \\
&[A_o \supset B_o] \to [[\sim A_o] \vee B_o] \\
&[A_o \equiv B_o] \to [[A_o \supset B_o][B_o \supset A_o]] \\
&[(x_\alpha A_o)] \to \Pi_{o(o\alpha)} (\lambda x_\alpha A_o) \\
&[(\exists x_\alpha A_o)] \to [\sim [(x_\alpha)[\sim A_o]]] \\
&[(, x_\alpha A_o)] \to \iota_{\alpha(o\alpha)} (\lambda x_\alpha A_o)
\end{aligned}
$$

Interpretação de cada linha

- $N : o \to o$ é um operador de negação, tal que $\neg A := NA : o$;

- $A : o \to o \to o$ é um operador de disjunção, tal que $A' \vee B' := AA'B' : o$
- A codificação do operador de conjunção é feito usando a lei de De Morgan, tal que $A \wedge B := \neg(\neg A \vee \neg B) : o$
- A codificação do operador de implicação é feito também em cima da disjunção $A \Rightarrow B := (\neg A) \vee B : o$
- A codificação da biimplicação é feito da forma normal, $A \Leftrightarrow B := (A \Rightarrow B) \wedge (B \Rightarrow A) : o$
- A codificação do quantificador universal é feito usando $\Pi : (\alpha \to o) \to o$, sendo $\forall(x : \alpha)A := \Pi(\lambda x : \alpha.A) : o$
- A codificação do quantificador existencial é feito a partir do quantificador universal $\exists(x : \alpha)A := \neg(\forall(x : \alpha)\neg A) : o$
- O último quantificador é um operador de seleção próximo ao axioma da escolha, que pega um termo de cada tipo, construido no Principia Mathematica

A relação de igualdade entre dois termos é construido dentro de o, usando Q:

$$
\begin{aligned}
&Q_{o\alpha\alpha} \to \lambda x_\alpha \lambda y_\alpha [(f_{o\alpha})[f_{o\alpha}x_\alpha \supset f_{o\alpha}y_\alpha]] \\
&[A_\alpha = B_\alpha] \to Q_{o\alpha\alpha}A_\alpha B_\alpha \\
&[A_\alpha \neq B_\alpha] \to [\sim (A_\alpha = B_\alpha)]
\end{aligned}
$$

Interpretação de cada linha:

- $Q : \alpha \to \alpha \to o$ é um termo que para dois termos de tipo α ele retorna a proposição que mapeia todas as funções que são válidas para ambos os termos
- $A = B := QAB : o$
- $A \neq B := \neg(A = B) : o$

A construção dos combinadores **I** e **K** são os mesmos:

$$
\begin{aligned}
&I_{\alpha\alpha} \to \lambda x_\alpha x_\alpha \\
&K_{\alpha\beta\alpha} \to \lambda x_\alpha \lambda y_\beta x_\alpha
\end{aligned}
$$

Uma notação adotada por Church é usar α' como $((\alpha\alpha)(\alpha\alpha))$, ou $(\alpha \to \alpha) \to \alpha \to \alpha$, α'' como $((\alpha'\alpha')(\alpha'\alpha'))$, etc.
A codificação dos numerais é a, assim chamada codificação de Church:

$0_{\alpha'} \to \lambda f_{\alpha\alpha}\lambda x_\alpha x_\alpha$
$1_{\alpha'} \to \lambda f_{\alpha\alpha}\lambda x_\alpha (f_{\alpha\alpha} x_\alpha)$
$2_{\alpha'} \to \lambda f_{\alpha\alpha}\lambda x_\alpha (f_{\alpha\alpha}(f_{\alpha\alpha} x_\alpha))$
$3_{\alpha'} \to \lambda f_{\alpha\alpha}\lambda x_\alpha (f_{\alpha\alpha}(f_{\alpha\alpha}(f_{\alpha\alpha} x_\alpha)))$
...
$S_{\alpha'\alpha'} \to \lambda n_{\alpha'}\lambda f_{\alpha\alpha}\lambda x_\alpha f_{\alpha\alpha}(n_{\alpha'} f_{\alpha\alpha} x_\alpha)$
$N_{o\alpha'} \to \lambda n_{\alpha'}[(f_{o\alpha'}) f_{o\alpha'} 0_{\alpha'} \supset [[(x_{\alpha'})[f_{o\alpha'} x_{\alpha'} \supset f_{o\alpha'}(S_{\alpha'\alpha} x_{\alpha'})]] \supset f_{o\alpha'} n_{\alpha'}]]$
$\omega_{\alpha''\alpha'\alpha'} \to \lambda y_{\alpha'}\lambda z_{\alpha'}\lambda f_{\alpha'\alpha'}\lambda g_{\alpha'}\lambda h_{\alpha\alpha}\lambda x_\alpha (y_{\alpha'}(f_{\alpha'\alpha'} g_{\alpha'} h_{\alpha\alpha})(z_{\alpha'}(g_{\alpha'} h_{\alpha\alpha}) x_\alpha))$
$\langle A_{\alpha'}, B_{\alpha'}\rangle \to \omega_{\alpha''\alpha'\alpha'} A_{\alpha'} B_{\alpha'}$
$P_{\alpha'\alpha'''} \to \lambda n_{\alpha'''}(n_{\alpha'''}(\lambda p_{\alpha''}\langle S_{\alpha'\alpha'}(p_{\alpha''}(K_{\alpha'\alpha'\alpha'} I_{\alpha'}) 0_{\alpha'}), p_{\alpha''}(K_{\alpha'\alpha'\alpha'} I_{\alpha'}) 0_{\alpha'}\rangle)\langle 0_{\alpha'}, 0_{\alpha'}\rangle, (K_{\alpha'\alpha'\alpha'} 0_{\alpha'}) I_{\alpha'})$
$T_{\alpha''\alpha'} \to \lambda x_{\alpha'}[(, x_{\alpha''})[(N_{o\alpha''} x_{\alpha''})[x_{\alpha''} S_{\alpha'\alpha'} 0_{\alpha'} = x_{\alpha'}]]]$
$P_{\alpha'\alpha'} \to \lambda x_{\alpha'}(P_{\alpha'\alpha'''}(T_{\alpha'''a''}(T_{\alpha''a'} x_{\alpha'})))$

Interpretação

- Os números são os mesmos introduzidos na exposição da codificação de church no capítulo anterior, a diferença é que eles agora possuem o tipo α', que é o mesmo que $(\alpha \to \alpha) \to \alpha \to \alpha$
- $S : \alpha' \to \alpha'$ é a função sucessor, vista também anteriormente
- $N : \alpha' \to o$ denota a proposição "é um número natural (de tipo α')" (é equivalênte ao principio da indução)
- $T : \alpha' \to \alpha''$ transforma um número de tipo α' para outro de tipo α''
- $P : \alpha''' \to \alpha'$ é a função predecessor que pega um número de tipo α''' e retorna seu predecessor de tipo α'
- $P : \alpha' \to \alpha'$ é a função predecessor de mesmo tipo.

Um dos problemas de usar essa codificação na teoria dos tipos simples é que para cada tipo α existem números, S e N, existem duas formas de burlar isso de formas diferentes: ou usando a teoria dos tipos polimórfica (Isso será introduzido na exposição do Sistema F) ou tendo formas de transformar um número de um tipo em um número de outro tipo. Essa segunda forma é o caminho de Church, mas essa forma depende do axioma da escolha e não é construtivista.

III. Regras de Inferência

As regras de inferência introduzidas nesse capítulo são as seguintes:

> "As regras de inferência (ou regras de procedimento) são as seis seguintes:
>
> I Para substituir uma parte M_α de uma formula pelo resultado de substituir y_β por x_β por todo M_α, dado que x_β não é uma variável livre em M_α e y_β não ocorre em M_α.

II Para substituir qualquer parte $((\lambda_\beta M_\alpha)N_\beta)$ de uma formula com o resultado de substituir N_β por x_β por todo M_α, dado que as variáveis ligadas de M_α são distintas tanto de x_β e das variáveis livres de N_β

III Onde A_α é o resultado de substituir N_β por x_β através de M_α, substituir qualquer parte A_α de uma formula por $((\lambda_\beta M_\alpha)N_\beta)$, dado que as variáveis ligadas de M_α são distintas tanto de x_β quando das variáveis livres de N_β

IV De $F_{o\alpha}x_\alpha$ para inferir $F_{o\alpha}A_\alpha$, dado que x_α não é uma variável livre de $F_{o\alpha}$

V De $A_o \supset B_o$ e A_o, inferir B_o

VI De $F_{o\alpha}x_\alpha$ inferir $\Pi_{o(o\alpha)}F_{o\alpha}$ dado que x_α não é uma variável livre de $F_{o\alpha}$ " (CHURCH, 1940, pag. 60)

A regra I é equivalênte à α-conversão, a II e a III correspondem a β-redução, IV corresponde a α-conversão no tipo o, V é o modus ponens e VI é uma regra de introdução do quantificador universal

IV. Axiomas formais

Os axiomas formas são as seguintes formulas:

1. $p \vee p \supset p$
2. $p \supset p \vee q$
3. $p \vee q \supset q \vee p$
4. $p \supset q \supset (r \vee p \supset r \vee q)$

5^α $\Pi_{o(o\alpha)}f_{o\alpha} \supset f_{o\alpha}x_\alpha$

6^α $(x_\alpha)[p \vee f_{o\alpha}x_\alpha] \supset p \vee \Pi_{o(o\alpha)}f_{o\alpha}$

7 $(\exists x_\iota)(\exists y_\iota)x_\iota \neq y_\iota$

8 $N_{o\iota'}x_{\iota'} \supset N_{o\iota'}y_{\iota'} \supset (S_{\iota'\iota}x_{\iota'} = S_{\iota'\iota}y_{\iota'} \supset x_{\iota'} = y_{\iota'})$

9^α $f_{o\alpha}x_\alpha \supset (y_\alpha)[f_{o\alpha}y_\alpha \supset x_\alpha = y_\alpha] \supset f_{o\alpha}(\iota_{\alpha(o\alpha)}f_{o\alpha})$

$10^{\alpha\beta}$ $(x_\beta)[f_{\alpha\beta}x_\beta = g_{\alpha\beta}x_\beta] \supset f_{\alpha\beta} = g_{\alpha\beta}$

11^α $f_{o\alpha}x_\alpha \supset f_{o\alpha}(\iota_{\alpha(o\alpha)}f_{o\alpha})$

Interpretação:

Os axiomas 1-4 são suficientes para formar o cálculo proposicional e os axiomas 1-6^α formam o cálculo lógico funcional. Para construir a teoria dos números elementares é necessário adicionar a 1-6^α os axiomas $7, 8$ e 9^α. Para construir a análise real clássica é necessário usar $10^{\alpha\beta}$ (Axioma da extencionalidade) e 11^α (Axioma da escolha)

V. Teorema da dedução

VII - Teorema da dedução

- Regras de igualdade - 21 equivale a regra eta

VI. Postulados de Peano para aritmética

VIII -

VII. Propriedades de T

VIII. definição da recursão primitiva

3.3 Modelos da teoria dos tipos simples

Uma forma de dar significado às funções do cálculo λ foi utilizando modelos matemáticos. Como visto anteriormente nesse capítulo, outra forma, a que Church escolheu primeiramente, foi desenvolver uma teoria de tipos que podesse definir o domínio e contradomínio das funções do cálculo λ. Como visto, os modelos desenvolvidos para o cálculo λ não tipado ganham força com o modelo de Scott D_∞ que é construido em cima da teoria dos domínios e de CPOs.

Um problema que surge tendo tomado o STλC como a extensão que dava significado ao cálculo λ é que ele da significado *interno* ao cálculo, mas não possui força para desenvolver seu significado *externo*. Ou seja, é possível construir a matemática dentro da teoria dos tipos simples, mas não é possível definir a teoria dos tipos simples dentro dela mesma. Para isso, diversos modelos *semânticos* (ou seja, externos) foram construidos, alguns usando a Teoria das Categorias, como será visto na Parte III dessas notas, e outros usando intuições vindas dos modelos da teoria dos domínios de Scott.

O modelo definido nessa subseção será um apontado por (GIRARD; TAYLOR; LAFONT, 1989) que utiliza uma estrutura chamada de *espaços de coerência* (Coherence spaces, em inglês) ou *espaços coerêntes* (espace cohérent ou coherent space) para tratar dos tipos.

3.3.1 Espaços de Coerência

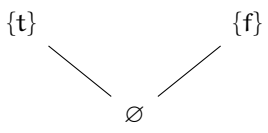
Primeiro, é necessário definir esses espaços de coerência:

Definição 3.13 (Espaços de Coerência, (GIRARD; TAYLOR; LAFONT, 1989))**.** Um *espaço de coerência* $\mathcal{A}$ é um conjunto que satisfaz:

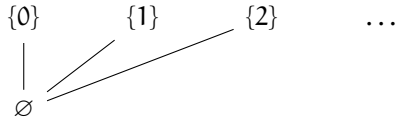
- (Fechamento abaixo) se $a \in \mathcal{A}$ e $a' \subset a$ então $a' \in \mathcal{A}$
- (Completude binária) se $M \subset \mathcal{A}$ e para todo $a_1, a_2 \in M$, com $a_1 \cup a_2 \in \mathcal{A}$, então $\bigcup M \in \mathcal{A}$

Em particular, tem-se que o conjunto vazio está em $\mathcal{A}$, $\varnothing \in \mathcal{A}$
Espaços de coerência podem ser considerados *algebricamente* como "domínios".
Exemplos:

- O espaço de coerência $\mathcal{B}\mathtt{ool}$ na forma:



- o espaço de coerência $\mathcal{I}\mathtt{nt}$ na forma:



Definição 3.14 ((GIRARD; TAYLOR; LAFONT, 1989))**.** Seja $|\mathcal{A}| \equiv \bigcup \mathcal{A} = \{\alpha : \{\alpha\} \in \mathcal{A}\}$. Os elementos de $|\mathcal{A}|$ são chamados de *tokens* e é a *relação de coerência modulo* $\mathcal{A}$ entre tokens é definida como:

$$\alpha \frown \alpha' \quad \text{iff} \quad \{\alpha, \alpha'\} \in \mathcal{A}$$

Essa relação é reflexiva e simétrica, então $|\mathcal{A}|$ equipada com $\frown$ é um grafo, chamado de *teia* de $|\mathcal{A}|$.

De uma teia é possível recuperar um espaço de coerência usando:

$$a \in \mathcal{A} \Leftrightarrow a \subset |\mathcal{A}| \wedge \forall \alpha_1, \alpha_2 \in a(\alpha_1 \frown \alpha_2 (\mathrm{mod} \mathcal{A}))$$

Logo, um ponto a de $\mathcal{A}$ forma um subgrafo completo, também chamado de um *clique*.

Na interpretação de modelo dos espaços de coerência, um tipo é equivalente a um espaço de coerência $\mathcal{A}$ e um termo desse tipo é um ponto em $\mathcal{A}$.

Para trabalhar com pontos de forma eficiente, é necessário introduzir uma noção de aproximação:

Definição 3.15 (Aproximante, (GIRARD; TAYLOR; LAFONT, 1989))**.** Um *aproximante* $a \in \mathcal{A}$ é qualquer subconjunto a' de a. Existem suficientes aproximantes finitos, ou seja:

- a é a união do conjunto de seus aproximantes finitos
- O conjunto I de aproximantes finitos é *direcionado*, ou seja:
  1. I não é vazio ($\varnothing \in \mathrm{I}$)
  2. Se $a', a'' \in \mathrm{I}$, é possível encontrar $a \in \mathrm{I}$ tal que $a', a'' \subset a$ (simplesmente $a = a' \cup a''$)

3.3.2 Funções Estáveis

Tendo definido o que são espaços de coerência, é importante definir como relacionar os espaços de coerência entre si:

Definição 3.16 (Função Estável, (GIRARD; TAYLOR; LAFONT, 1989))**.** Dados dois espaços de coerência $\mathcal{A}$ e $\mathcal{B}$, uma função $F : \mathcal{A} \to \mathcal{B}$ é dita *estável* se:

1. $a' \subset a \in \mathcal{A} \Rightarrow F(a') \subset F(a)$
2. (União direta) $F(\bigcup^{\uparrow}_{i \in I} a_i) = \bigcup^{\uparrow}_{i \in I} F(a_i)$
3. **(St)** $a_1 \cup a_2 \in \mathcal{A} \Rightarrow F(a_1 \cap a_2) = F(a_1) \cap F(a_2)$

A primeira condição diz que F preserva aproximações. Já a segunda condição mostra que F é contínua, ou seja

$$F(a) = \bigcup^{\uparrow} \{F(a_o) : a_o \subset a, a_o \text{ finite}\}$$

. A terceira condição é análoga à condição de estabilidade de Berry.

Essa definição se torna mais clara sendo tomada do ponto de vista categorial, onde $\mathcal{A}$ e $\mathcal{B}$ são categorias, pois F se torna um funtor que preserva limites diretos (união direta) e pullbacks (St).

Exemplos:

Algumas funções estáveis F de $\mathtt{Int}$ para $\mathtt{Int}$

- Função constante: se $F(\varnothing) = \{n\}$, então $F(a) = \{n\}$ para todo $a \in Int$
- Se $F(\varnothing) = \varnothing$, então considere a função parcial f definida exatamente nos inteiros n tais que $F(\{n\}) \neq \varnothing$, nesse caso $F(\{n\}) = \{f(n)\}$

Essas duas funções são importantes pois:

- No primeiro caso, tira-se as constantes $\dot{n}(a) = \{n\}$. Já no segundo
- No segundo caso, tira-se as funções $\tilde{f}(\varnothing) = \varnothing$, $\tilde{f}(\{m\}) = \{n\}$

É possível construir funções $F_n : Bool \to Bool \to Bool$ que se comportam como a disjunção, no caso $F_n(\{\alpha\}, \{\beta\}) = \{\alpha \vee \beta\}$ para cada combinação de verdadeiro ou falso de α e β.
$\{\alpha\}$ e $\{\beta\}$ podem possuir três valores: $\{t\}, \{f\}$ ou $\varnothing$, logo existem nove possibilidades:

- Os valores que se espera pela definição (onde nenhum dos dois é vazio) são:
  1. $F_1(\{t\}, \{t\}) = F_1(\{t\}, \{f\}) = F_1(\{f\}, \{t\}) = \{t\}$
  2. $F_1(\{f\}, \{f\}) = \{f\}$
- Para $F_1(\{t\}, \varnothing)$, $F_1(\{t\}, \varnothing) = \{t\}$ ou $F_1(\{t\}, \varnothing) = \varnothing$ é indiferente
- $F_1(\{f\}, \varnothing) = F_1(\varnothing, \{f\}) = \varnothing$, pois $F_1(\{f\}, \varnothing) \subset F_1(\{f\}, \{t\}) = \{t\}$ e $F_1(\varnothing, \{f\}) \subset F_1(\{t\}, \{f\}) = \{t\}$
- $F_1(\varnothing, \varnothing) = \varnothing$

Outra solução seria considerar $F_2(a, b) = F_1(b, a)$
A primeira solução pode ser enxugada da seguinte forma:

- $F_1(\{t\}, \{t\}) = F_1(\{t\}, \{f\}) = F_1(\{f\}, \{t\}) = \{t\}$
- $F_1(\{f\}, \{f\}) = \{f\}$
- $\varnothing$ caso contrário

3.3.3 Produto entre Espaços de coerência

Para definir a semântica do tipo produto, é necessário definir uma forma de relacionar dois espaços de coerência.

Para isso, é necessário primeiro generalizar a definição de função estável para funções com dois espaços de coerênia como domínio:

Definição 3.17. Uma função de dois argumentos, mapeando $\mathcal{A}$ e $\mathcal{B}$ para $\mathcal{C}$ é *estável* quando:

1. Se $a' \subset a \in \mathcal{A}$ e $b' \subset b \in \mathcal{B}$, então $F(a', b') \subset F(a, b)$
2. $F(\bigcup^{\uparrow}_{i \in I} a_i, \bigcup^{\uparrow}_{j \in J} b_j) = \bigcup^{\uparrow}_{(i,j) \in I \times J} F(a_i, b_j)$
3. se $a_1 \cup a_2 \in \mathcal{A}$ e $b_1 \cup b_2 \in \mathcal{B}$, então $F(a_1 \cap a_2, b_1 \cap b_2) = F(a_1, b_1) \cap F(a_2, b_2)$

Também é possível generalizar essa definição para funções que recebem n argumentos (fica como exercício para o leitor). A terceira condição mostra que para que F seja estável em dois argumentos, F em cada argumento precisa ser estável por si só. Para evitar essa generalização, é interessante definir *produtos (diretos)* de espaços de coerência, denotados $\mathcal{A}\&\mathcal{B}$:

Definição 3.18 (Produto de dois espaços de coerência, (GIRARD; TAYLOR; LAFONT, 1989))**.** Se $\mathcal{A}$ e $\mathcal{B}$ são dois espaços de coerênica, então o produto deles $\mathcal{A}\&\mathcal{B}$ é dado por:

$$|\mathcal{A}\&\mathcal{B}| = |\mathcal{A}| + |\mathcal{B}| = \{1\} \times |\mathcal{A}| \cup \{2\} \times |\mathcal{B}|$$

Satisfazendo:

$$\begin{aligned}
&(1,\alpha) \frown (1,\alpha') \pmod{\mathcal{A}\&\mathcal{B}} \quad \text{iff } \alpha \frown \alpha' \pmod{\mathcal{A}}\\
&(2,\beta) \frown (2,\beta') \pmod{\mathcal{A}\&\mathcal{B}} \quad \text{iff } \beta \frown \beta' \pmod{\mathcal{B}}\\
&(1,\alpha) \frown (2,\beta) \pmod{\mathcal{A}\&\mathcal{B}} \quad \text{para todo } \alpha \in |\mathcal{A}| \text{ e } \beta \in |\mathcal{B}|
\end{aligned}$$

Os pontos de $\mathcal{A}\&\mathcal{B}$ podem ser escritos unicamente como $\{1\} \times a \cup \{2\} \times b$ com $a \in \mathcal{A}$ e $b \in \mathcal{B}$

Dada uma função F de $\mathcal{A}$ e $\mathcal{B}$ para $\mathcal{C}$, é possível definir uma função G de $\mathcal{A}\&\mathcal{B}$ para $\mathcal{C}$ como:

$$G(\{1\} \times a \cup \{2\} \times b) = F(a, b)$$

3.3.4 Espaço Funcional

Na perspectiva de tipos como espaços de coerência, o produto de espaços de coerência é equivalente ao tipo produto. Para definir o tipo funcional $A \to B$, seria necessario desenvolver um espaço de coerência que descreva o conjunto de funções estáveis $\mathcal{A} \to \mathcal{B}$. Para isso, é necessário definir o *traço* entre funções estáveis.

Lema 3.8 ((GIRARD; TAYLOR; LAFONT, 1989))**.** Seja F uma função estável de $\mathcal{A}$ para $\mathcal{B}$ e sejam $a \in \mathcal{A}$ e $\beta \in F(a)$, então:

- é possível encontrar $a_0 \subset a$ tal que $\beta \in F(a_0)$
- se a_0 é escolhido como o mínimo da inclusão entre as soluções de (i), então a_0 é o *menor* e é, particularmente, *único*.

Prova:

- Seja $a = \bigcup^{\uparrow}_{i \in I} a_i$, onde os a_i são subconjuntos finitos de a. Então $F(a) = \bigcup^{\uparrow}_{i \in I} F(a_i)$, e se $\beta \in F(a)$, $\beta \in F(a_{i_0})$ para algum $i_0 \in I$
- Seja a_0 mínimo e seja $a' \subset a$ tal que $\beta \in F(a')$. Então $(a_0 \cup a') \subset a \in \mathcal{A}$, então $a_0 \cup a' \in \mathcal{A}$ e $\beta \in F(a_0) \cap F(a') = F(a_0 \cap a')$. Como a_0 é mínimo, isso faz com que $a_0 \subset a_0 \cup a'$, então $a_0 \subset a'$ e a_0 é de fato o menor. □

Definição 3.19 (Traço, (GIRARD; TAYLOR; LAFONT, 1989))**.** O *traço* $\mathcal{T}r(F)$ é o conjunto de pares (a_0, β) tal que:

- a_0 é um ponto finito de $\mathcal{A}$ e $\beta \in |\mathcal{B}|$
- $\beta \in F(a_0)$
- se $a' \subset a_0$ e $\beta \in F(a')$, então $a' = a_0$

$\mathcal{T}r(F)$ determina F unicamente pela formula:

$$(\text{App}) \qquad F(a) := \{\beta | \exists a_0 \subset a (a_0, \beta) \in \mathcal{T}r(F)\}$$

Proposição 3.1 ((GIRARD; TAYLOR; LAFONT, 1989))**.** Enquanto F varia entre as funções estáveis de $\mathcal{A}$ a $\mathcal{B}$, seus traços dão pntos em um espaço de coerência, escrito $\mathcal{A} \to \mathcal{B}$

Prova: Seja o espaço de coerência $\mathcal{C}$ dado por $|\mathcal{C}| = \mathcal{A}_{\mathtt{fin}} \times |\mathcal{B}|$ ($\mathcal{A}_{\mathtt{fin}}$ é o conjunto de pontos finitos de $\mathcal{A}$) onde $(a_1, \beta_1) \frown (a_2, \beta_2) \pmod{\mathcal{C}}$ se:

- $a_1 \cup a_2 \in \mathcal{A} \Rightarrow \beta_1 \frown \beta_2 (\text{mod } \mathcal{B})$
- $a_1 \cup a_2 \in \mathcal{A} \wedge a_1 \neq a_2 \Rightarrow \beta_1 \neq \beta_2 (\text{mod } \mathcal{B})$

Se F é estável, então $\mathcal{T}r(F)$ é um subconjunto de $|\mathcal{C}|$ por construção. É necessário verificar o modulo de coerência em $\mathcal{C}$ de (a_1, β_1) e $(a_2, \beta_2) \in \mathcal{T}r(F)$:

- Se $a_1 \cup a_2 \in \mathcal{A}$, então $\{\beta_1, \beta_2\} \subset F(a_1 \cup a_2)$ então $\beta_1 \frown \beta_2 (\text{mod } \mathcal{B})$
- Se $\beta_1 = \beta_2$ e $a_1 \cup a_2 \in \mathcal{A}$, então o lema aplicado a $\beta_1 \in F(a_1 \cup a_2)$ dá que $a_1 = a_2$

Por outro lado, seja f um ponto de $\mathcal{C}$. A função entre $\mathcal{A}$ e $\mathcal{B}$ é definida pela formula:

$$(\text{App}) \qquad F(a) := \{\beta | \exists a_0 \subset a (a_0, \beta) \in f\}$$

Dessa forma:

- F é monótona
- Se $a = \bigcup^{\uparrow}_{i \in I} a_i$, então $F(a) = \bigcup^{\uparrow}_{i \in I} F(a_i)$ por monotonicidade. Por outro lado, se $\beta \in F(a)$, então existe um $a' \subset a$ finito tal que $\beta \in F(a')$, mas como $a' \subset \bigcup^{\uparrow}_{i \in I} a_i$, tem-se que $a' \subset a_k$ para algum k, então $\beta \in F(a_k)$ e a inclusão contrária é estabelecida
- Se $a_1 \cup a_2 \in \mathcal{A}$, então $F(a_1 \cap a_2) \subset F(a_1) \cap F(a_2)$ por monotonicidade. De forma contrária, se $\beta \in F(a_1) \cap F(a_2)$, então $(a'_1, \beta), (a'_2, \beta) \in f$ para alguns $a'_1 \subset a_1$ e $a'_2 \subset a_2$ apropriados. Mas (a'_1, β) e (a'_2, β) são coerentes e $a'_1 \cup a'_2 \subset a_1 \cup a_2 \in \mathcal{A}$, então $a'_1 = a'_2$, com $a'_1 \subset a_1 \cap a_2$ e $\beta \in F(a_1 \cap a_2)$
- $F(a)$, para $a \in \mathcal{A}$, é um subconjunto de $|\mathcal{B}|$ e é necessário mostrar sua coerência. Agora, sejam $\beta', \beta'' \in F(a)$, isso significa que $(a', \beta'), (a'', \beta'') \in f$ para $a', a'' \subset a$ apropriados. Mas então $a' \subset a'' \subset a \in \mathcal{A}$, então (a', β') e (a'', β'') são coerentes, $\beta' \frown \beta'' (\text{mod } \mathcal{B})$ $\square$

Sendo $\mathcal{A} \to \mathcal{B}$ um espaço de coerência, ele é naturalmente ordenado por inclusão, pela seguinte relação de ordem:

$$F \leq_B G \qquad \text{sse} \qquad \mathcal{T}r(F) \subset \mathcal{T}r(G)$$

Proposição 3.2. A *ordem de Berry* $\leq_B$ é dada por:

$$F \leq_B G \qquad \text{sse} \qquad \forall a, a' \in \mathcal{A}(a' \subset a \Rightarrow F(a') = F(a) \cap G(a'))$$

Prova: Se $F \leq_B G$, então $F(a) \subset G(a)$ para todo a. Seja $(a, \beta) \in \mathcal{T}r(F)$, então $\beta \in F(a) \subset G(a)$. É necessário provar que $(a, \beta) \in \mathcal{T}r(G)$. Seja $a' \subset a$ tal que $\beta \in G(a')$, então $\beta \in F(a) \cap G(a') = F(a')$, que faz com que $a' = a$

Por outro lado, se $\mathcal{T}r(F) \subset \mathcal{T}r(G)$, é fácil ver que $F(a) \subset G(a)$ para todo a. De forma particular se $a' \subset a$, então $F(a') \subset F(a) \cap G(a')$. Agora se $\beta \in F(a) \cap G(a')$, é possível encontrar $a_0 \subset a$ e $a'_0 \subset a_0$ tais que:

$$(a_0, \beta) \in \mathcal{T}r(F)$$

e

$$(a'_0, \beta) \in \mathcal{T}r(G)$$

Então (a_0, β) e (a'_0, β) são coerentes, e como $a_0 \cup a'_0 \subset a \in \mathcal{A}$, tem-se que $a_0 = a'_0$ e $\beta \in F(a'_0) = F(a_0) \subset F(a')$ $\square$

3.3.5 A semântica do Tipo Soma

3.3.6 linearização

3.3.7 A semântica linearizada do tipo soma

3.3.8 A semântica do tipo unitário e Produtos Tensoriais

3.3.9 A semântica denotacional da teoria dos tipos simples

4 O Sistema F

No Cálculo-Lambda Simplesmente Tipado, é possível definir a função identidade, a função que pega um valor como input e retorna o próprio valor como outpu, para cada tipo definido no cálculo:

- Para os números naturais, $\lambda x : \mathbb{N}.x$
- Para os booleanos, $\lambda x : \mathtt{bool}.x$
- Para o tipo das funções dos naturais nos booleanos, $\lambda x : (\mathbb{N} \to \mathtt{bool}).x$
- ...

Mas dessa forma, quanto mais tipos a teoria suportar, mais formais diferentes são possíveis de serem criadas. Isso faz com que existam vários termos análogos sem qualquer possibilidade de relação entre eles. O máximo que se pode dizer é fazer uma quantificação além de $\lambda_{\to}$ e construir um tipo arbitrário α com uma função $f \equiv \lambda x : \alpha.x$ que seria a função identidade arbitrária.

Porém, dado um termo $M : \mathbb{N}$, não é possível escrever fM pois $\alpha \not\equiv \mathbb{N}$. Para fazer isso, é necessário que a função receba também o tipo específico que ela precisa ter para receber o termo M, fazendo um segundo processo de abstração em cima da função da seguinte forma:

$$\lambda\alpha : *.\lambda x : \alpha.x$$

Nesse caso, α se torna uma variável de tipo e $\star$ o tipo de todos os tipos. Esse termo é chamado de *polimórfico*, pois pode possuir diversas formas diferentes a depender do tipo escolhido:

- $(\lambda\alpha : *.\lambda x : \alpha.x)\mathbb{N} \to_\beta \lambda x : \mathbb{N}.x$

Para fazer essa extensão, é necessário adicionar regras de inferência e regras de tipagem que lidem com essa abstração de segunda ordem.

A tipagem para a função identidade $\lambda\alpha : *.\lambda x : \alpha.x$ é o tipo $\Pi\alpha : *.\alpha \to \alpha$, onde Π é o operador que tem como função ligar os tipos, chamado de Tipo Π ou Tipo Produto

Exemplos:

- A função de iteração D que recebe uma função $f : \alpha \to \alpha$ e retorna a aplicação dela duas vezes em cima de um termo $x : \alpha$ pode ser escrita da seguinte forma:

  $$D \equiv \lambda\alpha : *.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f(fx)$$

  Nesse caso, D é a mesma coisa que $f \circ f$. Para os números naturais:

  $$D\mathbb{N} \equiv \lambda f : \mathbb{N} \to \mathbb{N}.\lambda x : \mathbb{N}.f(fx)$$

  e sendo f a função sucessor s que mapeia $n : \mathbb{N}$ em $n + 1 : \mathbb{N}$, então:

  $$D\mathbb{N}s \to_\beta \lambda x : \mathbb{N}.s(sx)$$

  O tipo de D é: $D : \Pi\alpha : *.(\alpha \to \alpha) \to \alpha \to \alpha$

- A composição de duas funções é a aplicação de uma função em outra. É possível definir o operador de composição $\circ$ da seguinte forma:

$$\circ \equiv \lambda\alpha : *.\lambda\beta : *\lambda\gamma : *.\lambda f : \alpha \to \beta.\lambda g : \beta \to \gamma.\lambda x : \alpha.g(fx)$$

A sua tipagem é: $\circ : \Pi\alpha : *.\Pi\beta : *.\Pi\gamma : *.(\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$

4.1 O Cálculo Lambda com tipagem de Segunda Ordem

4.1.1 Regras de Inferência

Uma vez inseridas as regras de abstração e aplicação de segunda ordem, é necessário extender as regras de inferência em relação ao STλC

Definição 4.1 (Regra de Inferência para a Abstração)**.**

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash \lambda\alpha : *.M : \Pi\alpha : *.A}\ abst_2$$

Essa regra define basicamente que, sendo M um termo de tipo A em um contexto onde α possui tipo $*$, então a abstração $\alpha : *.M$ possui o tipo $\Pi\alpha : *.A$. Essa regra da abstração difere da primeira por permitir a definição de α no contexto.

Definição 4.2 (Regra de Inferência para a Aplicação)**.**

$$\frac{\Gamma \vdash M : \Pi\alpha : *.A \qquad \Gamma \vdash B : *}{\Gamma \vdash MB : A[\alpha := B]}\ appl_2$$

4.1.2 O Sistema $\lambda 2$

A sintaxe de $\lambda 2$ segue de forma análoga a λ_σ, sendo descrita pela seguinte BNF:

$$\mathbb{T}2 = \mathbb{V} | (\mathbb{T}2 \to \mathbb{T}2) | (\Pi\mathbb{V} : *.\mathbb{T}2)$$

onde $\mathbb{V}$ é a coleção dos tipos variáveis, denominados de $\alpha, \beta, \gamma, \ldots$.
Para os termos pré-tipados:

Definição 4.3. A coleção dos λ-termos pré-tipados de segunda ordem, ou λ2-termos, é definido na seguinte BNF:

$$\Lambda_{\mathbb{T}2} = V | (\Lambda_{\mathbb{T}2}\Lambda_{\mathbb{T}2}) | (\Lambda_{\mathbb{T}2}\mathbb{T}2) | (\lambda V : \mathbb{T}2.\Lambda_{\mathbb{T}2}) | (\lambda\mathbb{V} : *.\Lambda_{\mathbb{T}2})$$

Onde V é a coleção das variáveis de termos $(x, y, z, \ldots)$. Como existem ambos $\mathbb{V}$ e V, então a BNF possui duas formas de aplicação, uma de primeira ordem $(\lambda V : \mathbb{T}2.\Lambda_{\mathbb{T}2})$ para variáveis de termo e outro de segunda ordem $(\lambda\mathbb{V} : *.\Lambda_{\mathbb{T}2})$ para variáveis de tipo.

Da mesma forma, também existe a aplicação de primeira ordem $(\Lambda_{\mathbb{T}2}\Lambda_{\mathbb{T}2})$ e de segunda ordem $(\Lambda_{\mathbb{T}2}\mathbb{T}2)$.

As regras de parenteses em aplicação e abstração segue as regras vistas anteriormente para o STλC e para o $\lambda_{\beta\eta}$:

- Parenteses mais externos podem ser omitidos
- Aplicação é associativa à esquerda

- Aplicação e $\rightarrow$ precedem ambas abstrações λ e Π
- Abstrações λ e Π sucessivas com o mesmo tipo podem ser combinadas de forma associativa à direita
- Tipos funcionais são escritos de forma associativa à direita

Exemplo: $(\Pi\alpha : *.(\Pi\beta : *.(\alpha \rightarrow (\beta \rightarrow \alpha))))$ pode ser escrito como $\Pi\alpha, \beta : *.\alpha \rightarrow \beta \rightarrow \alpha$.

A definição para declarações e sentenças pode ser estendida da seguinte forma:

Definição 4.4 (Declarações, sentenças)**.**

- Uma *sentença* possui a forma $M : \sigma$ onde $M \in \Lambda_{\mathbb{T}2}$ e $\sigma \in \mathbb{T}2$ ou da forma $\sigma : *$, onde $\sigma \in \mathbb{T}2$
- Uma *declaração* é uma sentença com uma variável de termo ou uma variável de tipo como sujeito

Para $\lambda 2$ como é possível que uma variável de termo faça uso de uma variável de tipo, é necessário que a ordem da aparição dessas variáveis siga uma regra, para que uma variável não seja usada antes de ser declarada. O contexto pode ser descrito como um *domínio* da seguinte forma:

Definição 4.5 (Contexto de $\lambda 2$)**.**

1. $\emptyset$ é um contexto válido de $\lambda 2$
   $\mathrm{dom}(\emptyset) = ()$, a lista vazia
2. Se Γ for um contexto de $\lambda 2$, $\alpha \in \mathbb{V}$ e $\alpha \notin \mathrm{dom}(\Gamma)$, então $\Gamma, \alpha : *$ é um contexto de $\lambda 2$
   $\mathrm{dom}(\Gamma, \alpha : *) = (\mathrm{dom}(\Gamma), \alpha)$, ou seja $\mathrm{dom}(\Gamma)$ concatenado com α
3. Se Γ for um contexto de $\lambda 2$, se $\rho \in \mathbb{T}2$ tal que $\alpha \in \mathrm{dom}(\Gamma)$ para toda variável de tipo livre α existente em ρ e se $x \notin \mathrm{dom}(\Gamma)$, então $\Gamma, x : \rho$ é um contexto de $\lambda 2$
   $\mathrm{dom}(\Gamma, x : \rho) = (\mathrm{dom}(\Gamma), x)$

Exemplos

- $\emptyset$ é um contexto de $\lambda 2$ por (1)
- $\alpha : *$ é um contexto de $\lambda 2$ por (2)
- $\alpha : *, x : \alpha \rightarrow \alpha$ é um contexto de $\lambda 2$ por (3)
- logo $\alpha : *, x : \alpha \rightarrow \alpha, \beta : *$ é um contexto de $\lambda 2$ por (2)
- e $\alpha : *, x : \alpha \rightarrow \alpha, \beta : *, y : (\alpha \rightarrow \alpha) \rightarrow \beta$ é um contexto de $\lambda 2$ por (3), sendo $\mathrm{dom}(\Gamma) = (\alpha, x, \beta, y)$

A regra *var* pode ser reconstruida para lidar com os tipos de $\lambda 2$:

Definição 4.6. (Regra var em $\lambda 2$) (*var*) $\Gamma \vdash x : \sigma$ se Γ for um contexto de $\lambda 2$ e $x : \sigma \in \Gamma$

O problema é que, usando as regras até então, não é possível chegar ao juizo $\Gamma \vdash B : *$. Por isso, será introduzida uma nova regra:

Definição 4.7. (Regra de formação) (*form*) $\Gamma \vdash B : *$ se Γ é um contexto de $\lambda 2$, $B \in \mathbb{T}2$ e todas as variáveis de tipo livres em B sejam declaradas em Γ

Regras de $\lambda 2$:

- (*var*) $\Gamma \vdash x : \sigma$ se Γ for um contexto de $\lambda 2$ e $x : \sigma \in \Gamma$
- (*appl*)

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ \text{appl}$$

- (*abst*)

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma . M : \sigma \to \tau}\ \text{abst}$$

- (*form*) $\Gamma \vdash B : *$ se Γ é um contexto de $\lambda 2$, $B \in \mathbb{T}2$ e todas as variáveis de tipo livres em B sejam declaradas em Γ
- $(\mathtt{appl}_2)$

$$\frac{\Gamma \vdash M : \Pi \alpha : * . A \qquad \Gamma \vdash B : *}{\Gamma \vdash MB : A[\alpha := B]}\ \mathtt{appl}_2$$

- $(\mathtt{abst}_2)$

$$\frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash \lambda \alpha : * . M : \Pi \alpha : * . A}\ \mathtt{abst}_2$$

Definição 4.8. ($\lambda 2$-termos legais) Um termo M em $\Lambda_{\mathbb{T}2}$ é chamado de *legal* se existe um contexto de $\lambda 2$ Γ e um tipo ρ em $\mathbb{T}2$ tal que $\Gamma \vdash M : \rho$

4.1.3 Exemplos de Derivação

Seja a seguinte árvore de inferência incompleta:

$$\frac{?}{\emptyset \vdash \lambda \alpha : * . \lambda f : \alpha \to \alpha . \lambda x : \alpha . f(fx) : \Pi \alpha : * . (\alpha \to \alpha) \to \alpha \to \alpha}\ ?$$

Primeiro, é necessário utilizar a regra $(\mathtt{abst}_2)$:

$$\frac{\dfrac{?}{\alpha : * \vdash \lambda f : \alpha \to \alpha . \lambda x : \alpha . f(fx) : (\alpha \to \alpha) \to \alpha \to \alpha}\ ?}{\emptyset \vdash \lambda \alpha : * . \lambda f : \alpha \to \alpha . \lambda x : \alpha . f(fx) : \Pi \alpha : * . (\alpha \to \alpha) \to \alpha \to \alpha}\ \mathtt{abst}_2$$

Após isso as regras que precisam ser utilizadas já são conhecidas a partir do STλC:

primeiro dois absts seguidos para f e x:

$$\frac{\dfrac{\dfrac{?}{\alpha : *, f : \alpha \to \alpha, x : \alpha \vdash f(fx) : \alpha}\ ?}{\alpha : *, f : \alpha \to \alpha \vdash \lambda x : \alpha . f(fx) : \alpha \to \alpha}\ \mathtt{abst}}{\alpha : * \vdash \lambda f : \alpha \to \alpha . \lambda x : \alpha . f(fx) : (\alpha \to \alpha) \to \alpha \to \alpha}\ \mathtt{abst}}{\emptyset \vdash \lambda \alpha : * . \lambda f : \alpha \to \alpha . \lambda x : \alpha . f(fx) : \Pi \alpha : * . (\alpha \to \alpha) \to \alpha \to \alpha}\ \mathtt{abst}_2$$

O resto da Derivação fica como exercício para o leitor

4.1.4 Propriedades de $\lambda 2$

A definição de α-conversão deve ser acomodada para lidar com tipos Π:

Definição 4.9 (α-conversão ou α-equivalência)**.**

1. (Renomeando variáveis de termo)
   $\lambda x : \sigma.M =_\alpha \lambda y : \sigma.M^{x \to y}$ se $y \notin FV(M)$ e y não ocorre como ligante em M

2. (Renomeando variáveis de tipo)
   $\lambda \alpha : *.M =_\alpha \lambda \beta : *.M[\alpha := \beta]$ se β não ocorre em M
   $\Pi \alpha : *.M =_\alpha \Pi \beta : *.M[\alpha := \beta]$ se β não ocorre em M

3. O resto das definições se segue da definição 1.8

Também é possível extender a regra de β-redução:

Definição 4.10. (β-redução de passo único)

1. (Base, de primeira ordem) $(\lambda : \sigma.M)N \to_\beta M[x := N]$

2. (Base, de segunda ordem) $(\lambda \alpha : *.M)T \to_\beta M[\alpha := T]$

3. (Compatibilidade) da mesma forma que definição 1.10

Os lemmas definidos no capítulo 2 também podem ser utilizados aqui:

Lema 4.1. Os seguintes lemas e teoremas também são válidos para $\lambda 2$:

- Lema das variáveis livres
- Lema do afinamento
- Lema da condensação
- Lema da geração
- Lema do subtermo
- Unicidade dos tipos
- Lema da substituição
- Teorema de Church-Rosser
- Redução do sujeito
- Teorema da normalização forte

Lema 4.2 (Lema da permutação)**.** Se $\Gamma \vdash M : \sigma$ e Γ' é uma permutação de Γ e um contexto de $\lambda 2$ válido, então Γ' também é um contexto e $\Gamma' \vdash M : \sigma$.

4.2 O Sistema $\mathcal{F}$ de girard

O sistema $\mathcal{F}$ de Girard possui tipagem igual a tipagem na subseção anterior, com tipos Π e abstração em tipos.

Em (GIRARD; TAYLOR; LAFONT, 1989), Girard utiliza a notação de Λ no lugar do λ quando for feita abstração no tipo. Para exemplificar, o termo que dado um tipo gera uma função identidade é a seguinte:

$$\lambda\alpha : *.\lambda x : \alpha.x : \Pi\alpha : *.\alpha \to \alpha$$

Na notação de (GIRARD; TAYLOR; LAFONT, 1989):

$$\Lambda\alpha.\lambda x^{\alpha}.x^{\alpha} : \Pi\alpha.\alpha \to \alpha$$

Alguns pontos:

- O λ que abstrai o tipo se torna Λ
- A notação de tipagem do termo vira um superíndice
- A notação que α é um tipo não é utilizada no tipo

Girard aponta que o sistema $\mathcal{F}$ cobre problemas encontrados no Sistema T de Gödel, pois enquanto último postula tipos, o outro constrói os tipos em cima de termos, realizando a codificação de Church.

Nessa subseção, serão expostos esses tipos na forma de (GIRARD; TAYLOR; LAFONT, 1989).

4.2.1 Tipos Simples

I. Booleanos

O tipo dos booleanos é definido como:

$$\mathtt{Bool} \equiv \Pi X : *.X \to X \to X$$

Os termos T e F são termos que habitam o tipo $\mathtt{Bool}$:

$$\mathsf{T} \equiv \lambda X : *.\lambda x : X.\lambda y : Y.x \qquad \mathsf{F} \equiv \lambda X : *.\lambda x : X.\lambda y : Y.y$$

Sejam $u : U$, $v : U$ e $t : \mathtt{Bool}$, define-se $Duvt : U$ como:

$$Duvt \equiv tUuv$$

$Duvt$ é equivalente ao "if ... then ... else" presente em linguagens de programação, é possível atestar isso pois:

$$\begin{aligned} Duv\mathsf{T} &\equiv (\lambda X : *.\lambda x : X.\lambda y : X.x)Uuv \\ &\to_{\beta} (\lambda x : U.\lambda y : U.x)uv \\ &\to_{\beta} (\lambda y : U.u)v \\ &\to_{\beta} u \end{aligned}$$

e

$$\begin{aligned} Duv\mathsf{F} &\equiv (\lambda X : *.\lambda x : X.\lambda y : X.y)Uuv \\ &\to_\beta (\lambda x : U.\lambda y : U.y)uv \\ &\to_\beta (\lambda y : U.y)v \\ &\to_\beta v \end{aligned}$$

A função $neg(b)$ que leva de T para F e de F para T pode ser construida da seguinte forma: $neg(b) \equiv DFTb$

II. Produtos de Tipos
Dados dois tipos U e V, o seu produto $U \times V$ é definido como:

$$U \times V \equiv \Pi X : *.(U \to V \to X) \to X$$

Seja $u : U$ e $v : V$, o termo de tipo $U \times V$ é:

$$\langle u, v\rangle \equiv \lambda X : *.\lambda x : U \to V \to X.xuv$$

Suas projeções são definidas como:

$$\pi^1 t \equiv tU(\lambda x : U.\lambda y : V.x) \qquad \pi^2 t \equiv tV(\lambda x : U.\lambda y : V.y)$$

Para provar isso, será cálculado $\pi^1 \langle u, v\rangle$ e $\pi^2 \langle u, v\rangle$:

$$\begin{aligned} \pi^1 \langle u, v\rangle &\equiv (\lambda X : *.\lambda x : U \to V \to X.xuv)U(\lambda x : U.\lambda y : V.x) \\ &\to_\beta (\lambda x : U \to V \to U.xuv)(\lambda x : U.\lambda y : V.x) \\ &\to_\beta (\lambda x : U.\lambda y : V.x)uv \\ &\to_\beta (\lambda y : V.u)v \\ &\to_\beta u \end{aligned}$$

$$\begin{aligned} \pi^2 \langle u, v\rangle &\equiv (\lambda X : *.\lambda x : U \to V \to X.xuv)V(\lambda x : U.\lambda y : V.y) \\ &\to_\beta (\lambda x : U \to V \to V.xuv)(\lambda x : U.\lambda y : V.y) \\ &\to_\beta (\lambda x : U.\lambda y : V.y)uv \\ &\to_\beta (\lambda y : V.y)v \\ &\to_\beta v \end{aligned}$$

Um problema é que $\langle \pi^1 t, \pi^2 t\rangle \to_\beta t$ não é uma redução válida.

III. Tipo Vazio
O tipo vazio é definido da seguinte forma:

$$Emp \equiv \Pi X : *.X$$

com $\epsilon_U t \equiv tU$

Esse tipo é escolhido como tipo vazio pois não é habitado no Sistema $\mathcal{F}$

IV. Tipo Soma

Sejam U e V, é possível construir o tipo de soma disjunta da seguinte forma:

$$U + V \equiv \Pi X : *.(U \to X) \to (V \to X) \to X$$

Seja $u : U$ e $v : V$, é possível definir $\iota^1 u$ e $\iota^2 u$ de tipo $U + V$ da seguinte forma:

$$\iota^1 u \equiv \lambda X : *.\lambda x : U \to V.\lambda y : V \to X.xu \qquad \iota^2 v \equiv \lambda X : *.\lambda x : U \to V.\lambda y : V \to X.yv$$

Sejam $u : U$, $v : U$ e $t : R + S$, então é possível definir $\delta xuyvt$ de tipo U como:

$$\delta xuyvt \equiv tU(\lambda x : U.u)(\lambda y : V.v)$$

Calculando $\delta xuyv(\iota^1 r)$:

$$\begin{aligned}
\delta xuyv(\iota^1 r) &\equiv (\lambda X : *.\lambda x : R \to X.\lambda y : S \to X.xr)U(\lambda x : R.u)(\lambda y : S.v) \\
&\to_\beta (\lambda x : R \to U.\lambda y : S \to U.xr)(\lambda x : R.u)(\lambda y : S.v) \\
&\to_\beta (\lambda y : S \to U.(\lambda x : R.u)r)(\lambda y : S.v) \\
&\to_\beta (\lambda x : R.u)r \\
&\to_\beta u[r/x]
\end{aligned}$$

de forma similar $\delta xuyv(\iota^2 s) \to_\beta v[s/y]$

V. Tipo Existencial

Seja V um tipo e X uma variável de tipo, então:

$$\Sigma X.V \equiv \Pi Y : *.(\Pi X : *.(V \to X)) \to Y$$

Se U é um tipo e $v : V[U/X]$, então $\langle U, v\rangle$ é definido como de $\Sigma X.V$:

$$\langle U, v\rangle \equiv \lambda Y : *.\lambda x : (\Pi X : *.V \to Y).xUv$$

A eliminação de Σ é: se $w : W$ e $t : \Sigma X.V$, X sendo uma variável de tipo, x um termo de tipo V e as únicas ocorrências livres de X estão no tipo de x, é possível formar $\nabla Xxwt$ de tipo W:

$$\nabla Xxwt \equiv tW(\lambda X : *.\lambda x : V.w)$$

Calculando $(\nabla Xxwt)\langle U, v\rangle$:

$$
\begin{aligned}
(\nabla X x w t)\langle U, v\rangle &\equiv (\lambda Y : *.\lambda x : (\Pi X : *.V \to Y).xUv)W(\lambda X : *.\lambda x : V.w) \\
&\to_\beta (\lambda x : (\Pi X : *.V \to W).xUv)(\lambda X : *.\lambda x : V.w) \\
&\to_\beta (\lambda X : *.\lambda x : V.w)Uv \\
&\to_\beta (\lambda x : U.w)v \\
&\to_\beta w[U/X][v/x]
\end{aligned}
$$

4.2.2 Estruturas Livres

Para generalizar a construção de tipos em $\mathcal{F}$, é necessário definir estruturas livres:

Seja Θ uma coleção de expressões formais geradas por:

- alguns atomos $c_1, \ldots, c_k$
- algumas funções que permitam construir novos Θ-termos de termos antigos. O caso mais simples sãs as funções unárias de Θ para Θ, mas também é possível imaginar funções de vários argumentos de $\Theta, \Theta, \ldots, \Theta$ para Θ. Essas funções então possuem tipo $\Theta \to \Theta \to \cdots \to \Theta \to \Theta$. Incluindo o caso 0-ésimo (constantes), tem-se funções de n argumentos, com n possívelmente igual a 0

De forma geral:

Definição 4.11 ((GIRARD; TAYLOR; LAFONT, 1989))**.** A estrutura Θ será descrita por meio de um número finito de funções (*construtores*) $f_1, \ldots, f_n$ respectivamente de tipos $S_1, \ldots, S_n$. O tipo S_i deve por si mesmo ser da forma particular:

$$S_i = T_1^i \to T_2^i \to \cdots \to T_k^i \to \Theta$$

Com Θ ocorrendo somente de forma positiva em T_j^i

Todo elemento de Θ é representado de forma *única* por uma sucessão de aplicações de f_i.

Para representar Θ, o seguinte tipo é suficiente:

$$T = \Pi X : *.S_1 \to S_2 \to \cdots \to S_n \to X$$

Sejam $x_1, \ldots x_k$ argumentos de f_i, como X ocorre positivamente em T_j^i, a função canônica h_i de tipo $T \to X$ é definida por:

$$h_i x = xXy_1 \ldots y_n$$

com $y_i : S_i[T/X]$. Induz uma função $T_j^i[h_i]$ de $T_j^i[T/X]$ para T_j^i, dependente de $X, y_1, \ldots, y_n$. Essa função pode ser definida formalmente, mas não será feita aqui.

Finalmente,

$$f_i x_1 \ldots x_k = \lambda X : *.\lambda y_1 : S_1 \ldots \lambda y_n : S_n.y_i t_1 \ldots t_k$$

4.2.3 Tipos Indutivos

I. Inteiros

O que (GIRARD; TAYLOR; LAFONT, 1989) chama de inteiros, é o tipo dos números naturais, definidos por duas funções: O de tipo inteiro e S dos inteiros para inteiros, que dá $S_1 = X$ e $S_2 = X \to X$, então:

$$\mathrm{Int} \equiv \Pi X : *.X \to (X \to X) \to X$$

O inteiro n é representado por:

$$\bar{n} = \lambda X : *.\lambda x : X.\lambda y : X \to X.y(y(y \ldots (yx) \ldots))$$

Ao trocar S_1 por S_2, é possível desenvolver um tipo variante

$$\mathrm{Int}' \equiv \Pi X : *.(X \to X) \to (X \to X)$$

As funções básicas são:

$$O \equiv \lambda X : *.\lambda x : X.\lambda y : X \to X.x \qquad St \equiv \lambda X : *.\lambda x : X.\lambda y : X \to X.y(tXxy)$$

Logo $O = \bar{0}$ e $S\bar{n} \to_\beta \overline{n+1}$

O operador de indução é o *Iterador* It que pega um objeto de tipo U, uma função de tipo $U \to U$ e retorna um termo de tipo U.

$$\mathrm{Itu}ft = tUuf$$

Nos dois casos:

$$\begin{aligned}
\mathrm{Itu}fO &= (\lambda X : *.\lambda x : X.\lambda y : X \to X.x)Uuf \\
&\to_\beta (\lambda x : U.\lambda y : U \to U.x)uf \\
&\to_\beta (\lambda y : U \to U.u)f \\
&\to_\beta u \\
\mathrm{Itu}f(St) &= (\lambda X : *.\lambda x : X.\lambda y : X \to X.y(tXxy))Uuf \\
&\to_\beta (\lambda x : U.\lambda y : U \to U.y(tUxy))uf \\
&\to_\beta (\lambda y : U \to U.y(tUuy))f \\
&\to_\beta f(tUuf) \\
&\to_\beta f(\mathrm{Itu}ft)
\end{aligned}$$

Ou seja, o iterator repete a aplicação da função f $\bar{n}$ vezes. Mas não é verdadeiro que $\mathrm{Itu}f\overline{n+1} \to_\beta f(\mathrm{Itu}f\bar{n})$, mesmo que ambos sejam reduzíveis a $f(f(f \ldots (fu) \ldots))$.

Como o sistema $\mathcal{F}$ satisfaz a propriedade de Church-Rosser, é possível dizer que $\mathrm{Itu}f\overline{n+1}$ e $f(\mathrm{Itu}f\bar{n})$ são equivalentes, notado por $\mathrm{Itu}f\overline{n+1} \sim f(\mathrm{Itu}f\bar{n})$

Uma construção possível em cima da iteração é a recursão, para isso é necessário definir um termo auxiliar:

$$g = \lambda x : U \times \mathrm{Int}.\langle f(\pi^1 x)(\pi^2 x), S\pi^2 x\rangle$$

Em particular $g\langle u, \bar{n}\rangle \rightarrow_\beta \langle fu\bar{n}, \overline{n+1}\rangle$. Logo se $It\langle u, \bar{0}\rangle g\bar{n} \sim \langle t_n, \bar{n}\rangle$, então:

$$It\langle u, \bar{0}\rangle g\overline{n+1} \sim g(It\langle u, \bar{0}\rangle g\bar{n}) \sim g\langle t_n, \bar{n}\rangle \sim \langle ft_n\bar{n}, \overline{n+1}\rangle$$

Logo, é possível definir a recursão como: $Ruft \equiv \pi^1(It\langle u, \bar{0}\rangle gt)$, onde:

$$Ruf\bar{0} \sim u \qquad Ruf\overline{n+1} \sim f(Ruf\bar{n})\bar{n}$$

II. Listas

Seja U um tipo, seria interessante formar um tipo List$U$, que possui como objetos sequências finitas $(u_1, \dots u_n)$ de tipo U. Esse tipo possui duas funções:

- a sequência vazia $() :$ List$U$. Então $S_1 = X$
- a função que mapeia cada objeto $u : U$ e uma sequência $(u_1, \dots u_n)$ para a sequência $(u, u_1, \dots u_n)$. Então $S_2 = U \rightarrow X \rightarrow X$

Logo obtem-se:

$$\begin{aligned} \text{List } U &\equiv \Pi X : *.X \rightarrow (U \rightarrow X \rightarrow X) \rightarrow X \\ \text{nil} &\equiv \lambda X : *.\lambda x : X.\lambda y : U \rightarrow X \rightarrow X.x \\ \text{cons } ut &\equiv \lambda X : *.\lambda x : X.\lambda y : U \rightarrow X \rightarrow X.yu(tXxy) \end{aligned}$$

A sequência $(u_1, \dots u_n)$ é dada por:

$$\lambda X : *.\lambda x : X.\lambda y : U \rightarrow X \rightarrow X.yu_1(yu_2 \dots (yu_n x) \dots)$$

que é o mesmo que:

$$\lambda X : *.\lambda x : X.\lambda y : U \rightarrow X \rightarrow X.\text{cons } u_1(\text{cons } u_2 \dots (\text{cons } u_n \text{nil }) \dots)$$

Como a construção de lista é similar a dos números naturais, é possível construir um iterador de listas da seguinte forma:

Seja W um tipo, $w : W$, $f : U \rightarrow W \rightarrow W$, então para $t :$ List U o termo $Itwft : W$ é definida como:

$$Itwft \equiv tWwf$$

satisfazendo:

$$\text{It } wf \text{ nil } \rightarrow_\beta w \qquad \text{It } wf(\text{ Cons } ut) \rightarrow_\beta fu(\text{It } wft)$$

Exemplos:

- It nil cons $t \rightarrow_\beta t$ para todo t da forma $(u_1, \dots u_n)$
- Se $W =$ List V, onde V é um tipo outro, e $f = \lambda x : U.\lambda y :$ List W. cons $(gx)y$ onde $g : U \rightarrow V$, então:

$$\text{It nil } f(u_1, \dots u_n) \rightarrow_\beta (gu_1, \dots gu_n)$$

O leitor com maior familiaridade com programação funcional reconhecerá o segundo exemplo como a construção de *fold* em linguagens como Haskell
O recursor para listas é dado por:

$$\begin{aligned} \mathrm{R}vf\ \mathrm{nil} &\sim v \\ \mathrm{R}vf(u_1, \ldots u_n) &\sim fu_1(u_2, \ldots u_n)(\mathrm{R}vf(u_2, \ldots u_n)) \end{aligned}$$

III. Árvores Binárias
Árvores binárias finitas são construidas com duas funções:

- uma árvore consistindo de somente sua raiz, então $S_1 = X$
- uma árvore construida a partir de duas árvores, então $S_2 = X \to X \to X$

$$\begin{aligned} \text{Bintree} &\equiv \Pi X : *.X \to (X \to X \to X) \to X \\ \text{nil} &\equiv \lambda X : *.\lambda x : X.\lambda y : X \to X \to X.x \\ \text{couple}\ uv &\equiv \lambda X : *.\lambda x : X.\lambda y : X \to X \to X.y(uXxy)(vXxy) \end{aligned}$$

Iteração para as árvores binárias é dada por:

$$\mathrm{It}\ wft \equiv tWwf$$

com W sendo um tipo, $w : W$, $f : W \to W \to W$ e $t :$ Bintree. Sendo:

$$\mathrm{It}\ wf\ \mathrm{nil} \to_\beta w \qquad \mathrm{It}\ wf\ \text{couple}\ uv \to_\beta f(\mathrm{It}\ wfu)(\mathrm{It}\ wfv)$$

IV. Árvores de tipo U
Existem duas funções:

- a árvore consistindo de sua raiz, $S_1 = X$
- a construção de uma árvore de uma família $(t_u)_{u \in U}$ de árvores, então $S_2 = (U \to X) \to X$

$$\begin{aligned} \text{Tree}\ U &\equiv \Pi X : *.X \to ((U \to X) \to X) \to X \\ \text{nil} &\equiv \lambda X : *.\lambda x : X.\lambda y : (U \to X) \to X.x \\ \text{collect}\ f &\equiv \lambda X : *.\lambda x : X.\lambda y : (U \to X) \to X.y(\lambda z : U.fzXxy) \end{aligned}$$

O iterador (transfinito) é definido por:

$$\mathrm{It}\ wht \equiv tWwh$$

Para W sendo um tipo, $w : W$, $h : (U \to W) \to W$ e $t :$ Tree. Ele satisfaz:

$$\mathrm{It}\ wh\ \mathrm{nil} \to_\beta w \qquad \mathrm{It}\ wh(\ \text{collect}\ f) \to_\beta h(\lambda x : U.\ \mathrm{It}\ wh(fx))$$

4.3 Modelos do Sistema $\mathcal{F}$

5 A Teoria $\lambda\underline{\omega}$

5.1 A Teoria $\lambda\underline{\omega}$

Na seção anterior, foi introduzida a abstração em relação termos que podiam aceitar um tipo como parâmetro. Mas também é interessante construir tipos que aceitem tipos como parametros. Por exemplo, os tipos $\beta \to \beta$ e $\gamma \to \gamma$ possuem uma estrutura geral $\diamond \to \diamond$, com o tipo na mesma posição em relação à seta. Uma abstração em relação a $\diamond$ faz com que seja possível descrever uma família de tipos de forma mais simples.

Para isso, será introduzido aqui um *construtor de tipos* que gera uma função que recebe um tipo como valor e retorna um tipo como resultado, por exemplo $\lambda\alpha : *.\alpha \to \alpha$. Quando outros tipos são aplicados a essa função, ela muda seu comportamento:

$$(\lambda\alpha : *.\alpha \to \alpha)\beta \to_\beta \beta \to \beta$$

$$(\lambda\alpha : *.\alpha \to \alpha)\gamma \to_\beta \gamma \to \gamma$$

A questão que fica é definir o tipo dessas expressões. Pois sendo $\alpha : *$ e $\alpha \to \alpha : *$, então $\lambda\alpha : *.\alpha \to \alpha : * \to *$. Logo serão adicionados tipos como $* \to *$, $* \to (* \to *)$, etc. à sintaxe.

Os tipos $*$ e as setas entre $*$ são chamados de *espécies* (*kinds* em inglês). A BNF para o conjunto de todas as espécies é:

$$\mathbb{K} = *|\mathbb{K} \to \mathbb{K}$$

A notação dos parenteses segue a notação para os tipos simples introduzida anteriormente.

O tipo de todas as espécies é denotado por $\Box$. Sendo assim $* : \Box$ e $* \to * : \Box$, etc. Se κ é uma espécie, então qualquer termo M "do tipo" κ é chamado de construtor de tipos, ou somente *construtor*. Então $\alpha : *.\alpha \to \alpha$ é um construtor, assim como somente $\alpha \to \alpha$ também.

Definição 5.1 (Construtores, construtores próprios)**.**

1. Se $\kappa : \Box$ e $M : \kappa$, então M é um *construtor*. Se $\kappa \not\equiv *$, então M é um *construtor próprio*

2. O conjunto de todas as variedades (*sorts*) é $\{*, \Box\}$

Para falar de uma variedade qualquer, será introduzido o simbolo s como meta variável.

Definição 5.2 (níveis)**.** Com essa construção, existem quatro níveis na sintaxe: Nível 1: termos Nível 2: construtores e tipos com construtores próprios Nível 3: espécies Nível 4: consiste somente em $\Box$

Ao unir esses níveis é possível escrever correntes de juizos como $t : \sigma : * \to * : \Box$, onde $t : \sigma$, $\sigma : * \to *$ e $* \to * : \Box$ são juizos.

5.1.1 Regra sort e regra var em $\lambda\underline{\omega}$

É necessário escrever novas regras de inferência para $\lambda\underline{\omega}$, a primeira delas sendo a regra das espécies:

Definição 5.3 (Regra das variedades, Sort-rule)**.**
(*sort*) $\emptyset \vdash * : \Box$

A próxima regra é a regra de que todo termo ocorrendo em um contexto é derivável naquele contexto, para isso é necessário ter como base que o tipo do termo escolhido seja bem formado, então a regra (*var*) vai mudar em relação às teorias vistas anteriormente:

Definição 5.4 (Var-rule)**.**

$$(var)\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{se } x \notin \Gamma$$

A premissa dessa regra de derivação requer que A seja ou um tipo, se $s \equiv *$, ou uma espécie (se $s \equiv \Box$). Então x pode ser ou um tipo variável ou um termo variável.

Exemplo de derivação:

$$\frac{\dfrac{\emptyset \vdash * : \Box}{\alpha : * \vdash \alpha : *}\,(var)}{\alpha : *, x : \alpha \vdash x : \alpha}\,(var)$$

A primeira linha é formada utilizando a sort-rule, a segunda linha usa a var-rule com $s \equiv \Box$ e a terceira linha usa a var-rule com $s \equiv *$

5.1.2 A regra do enfraquecimento em $\lambda\underline{\omega}$

Somente usando as regras (*var*) e (*sort*) não é possível derivar $\alpha : *, \beta : * \vdash \alpha : *$, então é interessante desenvolver uma regra que permita fazer isso. A regra desejada seria uma regra que, partindo de $\alpha : * \vdash \alpha : *$, chegasse em $\alpha : *, \beta : * \vdash \alpha : *$. Ou seja, uma regra que adicionasse mais informação ao contexto do que o "necessario", que o enfraquecesse.

A regra do enfraquecimento segue a seguinte forma:

Definição 5.5 (Regra do enfraquecimento, (*weak*))**.**

$$(weak)\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \text{se } x \notin \Gamma$$

Ou seja, assumindo que tenha sido derivado o juizo $\Gamma \vdash A : B$, é possível enfraquecer o contexto Γ ao adicionar uma declaração arbitrária no final.

Então a derivação anterior se torna:

$$\frac{\dfrac{\emptyset \vdash * : \Box}{\alpha : * \vdash \alpha : *}\,(var) \qquad \dfrac{\dfrac{\emptyset \vdash * : \Box \qquad \emptyset \vdash * : \Box}{\alpha : * \vdash * : \Box}\,(weak)}{\alpha : *, \beta : * \vdash \alpha : *}\,(weak)$$

Também é possível fazer a seguinte derivação:

$$\frac{\dfrac{\emptyset \vdash * : \Box \qquad \emptyset \vdash * : \Box}{\alpha : * \vdash * : \Box}\,(weak)}{\alpha : *, \beta : * \vdash \beta : *}\,(var)$$

5.1.3 A regra de formação de $\lambda\underline{\omega}$

A regra de inferência para formar tipos e espécies é descrita como:

Definição 5.6 (Regra de Formação, (*form*))**.** $$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}\ (form)$$

Note que não existem tipos dependentes de tipos em $\lambda\underline{\omega}$, logo não existem tipos Π.

Exemplo:

$$\frac{\dfrac{\dots}{\alpha : *, \beta : * \vdash \alpha : *}\,(\dots) \qquad \dfrac{\dots}{\alpha : *, \beta : * \vdash \beta : *}\,(\dots)}{\alpha : *, \beta : * \vdash \alpha \to \beta : *}\ (form)$$

As duas subárvores geradas pela regra de formação nesse caso já foram detalhadas na subseção anterior, logo foram omitidas aqui.

Exemplo:

$$\frac{\dfrac{\dots}{\alpha : * \vdash * : \Box}\,(\dots) \qquad \dfrac{\dots}{\alpha : * \vdash * : \Box}\,(\dots)}{\alpha : * \vdash * \to * : \Box}\ (form)$$

5.1.4 Regras de abstração e aplicação

As regras de abstração e aplicação são definidas da seguinte forma:

Definição 5.7.

- (*appl*)

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\text{appl})$$

- (*abst*)

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}\ (\text{abst})$$

Exemplo: derivação de $(\lambda\alpha : *.\alpha \to \alpha)\beta$

$$\frac{\dfrac{?}{? \vdash \lambda\alpha : *.\alpha \to \alpha}\,? \qquad \dfrac{?}{? \vdash \beta : *}\,?}{? \vdash (\lambda\alpha : *.\alpha \to \alpha)\beta}\ (\text{appl})$$

A única regra que resolve o lado direito é a (*var*), logo o contexto deve ser também $\beta : *$:

$$\frac{\dfrac{?}{\beta : * \vdash \lambda\alpha : *.\alpha \to \alpha}\,? \qquad \dfrac{\emptyset \vdash * : \Box}{\beta : * \vdash \beta : *}\ (\text{var})}{\beta : * \vdash (\lambda\alpha : *.\alpha \to \alpha)\beta}\ (\text{appl})$$

Já no lado esquerdo, é necessário usar a regra (*abst*):

$$\frac{\dfrac{\beta : *, \alpha : * \vdash \alpha \to \alpha : * \qquad \beta : * \vdash * \to * : \Box}{\beta : * \vdash \lambda\alpha : *.\alpha \to \alpha}\ (\text{abst}) \qquad \dfrac{\emptyset \vdash * : \Box}{\beta : * \vdash \beta : *}\ (\text{var})}{\beta : * \vdash (\lambda\alpha : *.\alpha \to \alpha)\beta}\ (\text{appl})$$

O resto das duas subárvores do lado esquerdo se segue das derivações feitas anteriormente.

5.1.5 Regra da Conversão

A regra da conversão faz com que termos que possuem um tipo que possa ser β-reduzido a outro, possa passar a possuir o tipo mais simples:

Definição 5.8 (Regra de Conversão, (*form*))**.**

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \text{ se } B =_\beta B'$$

Regras de $\lambda\underline{\omega}$:

- (*sort*) $\emptyset \vdash * : \Box$
- (*var*)

$$(var)\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \text{ se } x \notin \Gamma$$

- (*weak*)

$$(weak)\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \text{ se } x \notin \Gamma$$

- (*form*)

$$\frac{\Gamma \vdash A : s \qquad \Gamma \vdash B : s}{\Gamma \vdash A \to B : s}\ (form)$$

- (*appl*)

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\text{appl})$$

- (*abst*)

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}\ (\text{abst})$$

- (*conv*)

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \text{ se } B =_\beta B'$$

5.1.6 Propriedades

O sistema $\lambda\underline{\omega}$ satisfaz a maioria das propriedades de sistemas anteriores. A única modificação necessária é no Lema da Unicidade dos tipos, pois tipos não são mais literalmente unicos, mas são únicos a menos de β-conversão:

Lema 5.1 (Unicidade dos tipos a menos de conversão)**.** Se $\Gamma \vdash A : B_1$ e $\Gamma \vdash A : B_2$, então $B_1 =_\beta B_2$

5.2 O Sistema $\mathcal{F}_\omega$ de Girard

6 Teoria dos Tipos Dependente

6.1 Teoria dos Tipos dependentes

Na Teoria dos Tipos simples $\lambda_{\rightarrow}$, cada termo depende de outro. Para cada extensão, foram adicionadas novas dependências:

- $\lambda 2$: termos dependem de tipos
- $\lambda\underline{\omega}$: tipos dependem de tipos

Fica faltando então uma teoria dos tipos que abarque tipos que dependem de termos.

- λP: tipos dependem de termos

É essa teoria que será analisáda nesse capítulo. Tipos que pendendem de termos possuem o seguinte formato:

$$\lambda x : A.M$$

onde M é um tipo e x é uma variável de termo (Logo A é um tipo também). A abstração $\lambda x : A.M$ *depende* do termo x.

Exemplos de Motivação:

(1) Na programação, podemos definir uma lista a partir de seu tamanho, por exemplo: $[1, 2] : \text{List}2$. Logo $\lambda n : \mathbb{N}.\text{List}n$ também é um tipo, também chamado de *construtor de tipo*, *família de tipos* ou *tipo indexado* (indexado pelo termo $n : \mathbb{N}$) que depende do termo n

(2) Seja $S_n = \{0, n, 2n, 3n, \dots\}$ o conjunto de todos os multiplos não negativos de n. Então $\lambda n : \mathbb{N}.S_n$ mapeia:

- $0 \mapsto \{0\}$
- $1 \mapsto \mathbb{N}$ (O conjunto de todos os números naturais)
- $2 \mapsto \{0, 2, 4, \dots\}$ (O conjunto de todos os números pares)

O tipo de S_n e de List$n$ é $\mathbb{N} \rightarrow *$.
Um exemplo importante é o seguinte:

(3) Seja P_n uma *proposição* para cada $n : \mathbb{N}$. A partir da interpretação de *Proposições-como-Tipos*, $\lambda n : \mathbb{N}.P_n$ é um tipo que mapeia n para sua proposição P_n correspondente, chamado de *função com valor de proposição*. Na lógica, esse tipo de construção é chamado de *Predicado*. Por exemplo, seja a interpretação de P_n como "n é um número primo". Na lógica, esse predicado pode ser verdadeiro ou falso a depender do valor de n

6.1.1 Regras de Inferência de λP

As regras de inferência de λP são as seguintes:

$$(sort)\ \emptyset \vdash * : \Box$$

$$(var)\ \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ \text{se } x \notin \Gamma$$

$$(weak)\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}\ \text{se } x \notin \Gamma$$

$$(form)\ \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

$$(appl)\ \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$(abst)\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$$

$$(conv)\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}\ \text{se } B =_\beta B'$$

As regras (*sort*), (*var*) e (*weak*) em λP são identicas às de $\lambda\underline{\omega}$. Porém, as regras que diferem de $\lambda\underline{\omega}$ são porque:

(i) Com o uso de tipos Π, nas regras de (*form*), (*appl*) e (*abst*) os tipos $\rightarrow$ não aparecem como em $A \rightarrow B$ e no lugar são colocados tipos como $\Pi x : A.B$.

(ii) No tipo $\Pi x : A.B$, x é necessariamente um *termo*, logo $A : *$. O que difere de $\lambda\underline{\omega}$

Em λP, a abstração só é escrita como $A \rightarrow B$ no lugar de $\Pi x : A.B$ se existe a certeza que x não ocorre livre em B.

Na regra (*form*), existem dois casos possíveis:

1. Se $s = *$, então $A : *$, $B : *$ e $\Pi x : A.B : *$
2. Se $s = \Box$, então $A : *$, $B : \Box$ e $\Pi x : A.B : \Box$

6.1.2 Exemplo de derivação em λP

Primeiro é interessante derivar o tipo $A \rightarrow * : \Box$:

$$\dfrac{(var)\ \dfrac{\emptyset \vdash * : \Box}{\emptyset \vdash A : *} \qquad (weak)\ \dfrac{\emptyset \vdash * : \Box \qquad (weak)\ \dfrac{\emptyset \vdash * : \Box \qquad (var)\ \dfrac{\emptyset \vdash * : \Box}{A : * \vdash * : \Box}}{\emptyset \vdash A : *}}{x : A \vdash * : \Box}}{(form)\ \dfrac{}{\emptyset \vdash A \rightarrow * : \Box}}$$

Uma vez construido esse tipo, é possível utilizar a regra (*var*) para gerar um habitante desse tipo:

$$(var)\ \frac{\emptyset \vdash A \rightarrow * : \Box}{P : A \rightarrow * \vdash P : A \rightarrow *}$$

Seja $x : A$ um termo, é possível construir a aplicação de P com x:

$$(appl)\ \frac{P : A \to * \vdash P : A \to * \qquad (var)\ \dfrac{\emptyset \vdash A : *}{x : A \vdash x : A}}{P : A \to *, x : A \vdash Px : *}$$

Podemos gerar o seguinte tipo:

$$(weak)\ \frac{P : A \to *, x : A \vdash Px : * \qquad P : A \to *, x : A \vdash Px : *}{P : A \to *, x : A, y : Px \vdash Px : *}$$

e

$$(form)\ \frac{P : A \to *, x : A \vdash Px : * \qquad P : A \to *, x : A, y : Px \vdash Px : *}{P : A \to *, x : A \vdash Px \to Px : *}$$

Logo:

$$(weak)\ \frac{\emptyset \vdash A : * \qquad \emptyset \vdash A \to * : \Box}{(form)\ \dfrac{P : A \to * \vdash A : * \qquad P : A \to *, x : A \vdash Px \to Px : *}{P : A \to * \vdash \Pi x : A.Px \to Px : *}}$$

Para gerar os termos:

$$(var)\ \frac{P : A \to *, x : A \vdash Px : *}{(abst)\ \dfrac{P : A \to *, x : A, y : Px \vdash y : Px \qquad P : A \to *, x : A \vdash Px \to Px : *}{(abst)\ \dfrac{P : A \to *, x : A \vdash \lambda y : Px.y : Px \to Px \qquad P : A \to * \vdash \Pi x : A.Px \to Px : *}{P : A \to * \vdash \lambda x : A \lambda y : Px.y : \Pi x : A.Px \to Px}}}$$

Fica para o leitor integrar essas díversas árvores em uma única.

6.1.3 Lógica de Predicados mínima em λP

Em λP é possível codificar uma forma de lógica simples chamada de *lógica de predicados mínima*. Essa lógica só possui a implicação e o quantificador universal em sua estrutura. As suas entidades básicas são *proposições*, *conjuntos* e *predicados sobre conjuntos*.

A interpretação de Proposições-como-Tipos (PAT) é feita da seguinte forma:

- Se o termo b habita o tipo B (ou seja, $b : B$) e sendo B interpretada como uma proposição, então b é a *prova* de B, chamado de *objeto de prova*.
- Se um tipo B não possui habitante, então não existe prova de B e B deve ser falso

Em λP, para definir que b habita B temos que realizar um juizo no estilo $\Gamma \vdash b : B$ a partir das regras de inferência descritas anteriormente.

Um conjunto S pode ser codificado como um tipo, então $S : *$. *Elementos* de um conjunto são termos. Então se a é um elemento de S, $a : S$. Se S for o conjunto vazio, S não vai possuir termos.

Exemplos: Se $\mathbb{N} : *$, $3 : \mathbb{N}$

Proposições também podem ser definidas como tipos. Então sendo A uma proposição, $A : *$. Um termo $p : A$ é uma prova de A.

Como visto anteriormente, um predicado P é uma função de um *conjunto* S para o *conjunto de todas as proposições*, então: $P : S \to *$. Logo seja P um predicado arbitrário em S, ou seja $P : S \to *$, então para cada $a : S$ tem-se $Pa : *$. Todo Pa é uma proposição, que é um tipo em λP, logo existem duas possibilidades:

1. Se Pa for *habitado*, ou seja existe $t : Pa$, então o predicado é válido para a
2. Caso Pa não seja habitado, o predicado não se segue para a

Anteriormente, foi identificada a implicação $A \Rightarrow B$ com o tipo $A \to B$ da seguinte forma:

$A \Rightarrow B$ é verdadeiro

Se A é verdadeiro, então B é verdadeiro

Se A é habitado, então B é habitado

Existe uma função mapeando habitantes de A em habitantes de B

Existe uma função $f : A \to B$

$A \to B$ é habitado

A partir das regras de λP é possível obter as regras de eliminação e introdução da implicação:

1. $\Rightarrow$-elim $\dfrac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$
2. $\Rightarrow$-intro $\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash A \to B : s}{\Gamma \vdash \lambda x : A.M : A \to B}$

Já a quantificação universal, $\forall_{x \in S} P(x)$, de um prediado P dependente de um x elemento de S vai ter sua equivalência encontrada da seguinte forma:

$\forall_{x \in S} P(x)$ é verdadeiro

Para cada x pertencente a S, a proposição $P(x)$ é verdadeira

para cada X em S, o tipo Px é habitado

Existe uma função mapeando cada x em S para um habitante de Px

Existe uma função f tal que $f : \Pi x : S.Px$

$\Pi x : S.Px$ é habitado

Logo, a forma de codificação de $\forall_{x \in S} P(x)$ é o tipo $\Pi x : S.Px$.
As regras de eliminação e introdução do $\forall$ no λP são as seguintes:

1. $\forall$-elim $\dfrac{\Gamma \vdash p : \forall_{x \in S} P(x) \qquad \Gamma \vdash n : S}{\Gamma \vdash pn : P(x)[x := n]}$
2. $\forall$-intro $\dfrac{\Gamma, x : S \vdash M : P(x) \qquad \Gamma \vdash \forall_{x \in S} P(x) : *}{\Gamma \vdash \lambda x : S.P(x) : \forall_{x \in S} P(x)}$

Essas regras correspondem, na dedução natural no estilo de Gentzen às seguintes:

1. $\forall I \dfrac{P(n)}{\forall_{x \in S} P(x)}$

2. $\forall E \dfrac{\forall_{x \in S} P(x)}{P(n)}$

6.1.4 Exemplo de derivação na lógica de predicados mínima

Seja S um conjunto de Q um predicado sobre S, então a seguinte proposição é provável usando a lógica de predicados mínima:

$$\forall_{x \in S} \forall_{y \in S} (Q(x, y)) \Rightarrow \forall_{u \in S} Q(u, u)$$

Na dedução natural, isso se torna:

$$\rightarrow I \dfrac{\forall E \dfrac{\forall E \dfrac{\forall E \dfrac{\forall_{x \in S} \forall_{y \in S} (Q(x, y))^1}{\forall_{y \in S} (Q(z, y))}}{Q(u, u)}}{\forall I \dfrac{}{\forall_{u \in S} Q(u, u)}}}{\forall_{x \in S} \forall_{y \in S} (Q(x, y)) \Rightarrow \forall_{u \in S} Q(u, u)}$$

Usando as regras de inferência introduzidas anteriormente:
Primeiro, é necessário traduzir essa proposição para um tipo:

$$\Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu$$

Então o problema se torna:

$$? \dfrac{?}{\emptyset \vdash ? : \Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu}$$

Usando as regras, é possível ver que o tipo $\Pi x : S.\Pi y : S.Qxy$ precisa ser definido por um termo único z na abstração:

$$\rightarrow\text{-intro} \dfrac{? \dfrac{?}{z : (\Pi x : S.\Pi y : S.Qxy) \vdash ? : \Pi u : S.Quu} \qquad \emptyset \vdash \Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu : *}{\emptyset \vdash \lambda z : (\Pi x : S.\Pi y : S.Qxy).? : \Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu}$$

Também por abstração, $\Pi u : S$ também se torna um termo próprio:

$$\rightarrow\text{-intro} \dfrac{\rightarrow\text{-intro} \dfrac{z : (\Pi x : S.\Pi y : S.Qxy), u : S \vdash ? : Quu \qquad z : (\Pi x : S.\Pi y : S.Qxy) \vdash S : *}{z : (\Pi x : S.\Pi y : S.Qxy) \vdash \lambda u : S.? : \Pi u : S.Quu}}{\emptyset \vdash \lambda z : (\Pi x : S.\Pi y : S.Qxy).\lambda u : S.? : \Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu}$$

A partir daqui é usado a regra da aplicação para o $\forall$:

$$\rightarrow\text{-intro}\ \frac{\forall\text{-elim}\ \dfrac{\forall\text{-elim}\ \dfrac{\rightarrow\text{-intro}\ \dfrac{z : (\Pi x : S.\Pi y : S.Qxy), u : S \vdash z : (\Pi x : S.\Pi y : S.Qxy)}{z : (\Pi x : S.\Pi y : S.Qxy), u : S \vdash zu : \Pi y : S.Quy}}{z : (\Pi x : S.\Pi y : S.Qxy), u : S \vdash zuu : Quu}}{z : (\Pi x : S.\Pi y : S.Qxy) \vdash \lambda u : S.zuu : \Pi u : S.Quu}}{\emptyset \vdash \lambda z : (\Pi x : S.\Pi y : S.Qxy).\lambda u : S.zuu : \Pi x : S.\Pi y : S.Qxy \rightarrow \Pi u : S.Quu}$$

O lado direto de cada passo é deixado para o leitor fazer por si só (Dica: usar uma folha A4 no modo paisagem).

Logo, o termo que prova que a proposição é verdadeira é o $\lambda z : (\Pi x : S.\Pi y : S.Qxy).\lambda u : S.zuu$. O interessante de descobrir o termo é que, somente a partir do termo, é possível reconstruir toda a prova novamente.

7 Calculo de Construções

7.1 O Cálculo de Construções e o λ-Cubo

7.1.1 O Sistema λC

O *Calculo de Construções* é a combinação das extenções do STλC vistas anteriormente: O λ2, o $\lambda\underline{\omega}$ e o λP. Esse sistema foi desenvolvido por Thierry Coquand em 1988. O C vem do seu sobrenome.

A única regra que difere λP de λC é a regra de formação, pois no λP, (*form*) se comporta da seguinte forma:

$$(\mathrm{form}_{\lambda P})\ \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

A primeira premissa garante que x seja um termo, ou seja que $\Pi x : A.B : s$ seja uma relação de tipos dependendo de termos. Porém em $\lambda\underline{\omega}$, tipos podem depender de tipos também, logo $A : *$ se torna $A : s$, onde s pode ser tanto $*$ ou $\Box$. Porém é necessário diferenciar o s da primeira premissa do s da seguinda premissa, logo serão usados os números 1 e 2 da seguinte forma:

$$(\mathrm{form}_{\lambda C})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$$

Logo essa regra de formação pode ter quatro formas possíveis a depender da escolha de s_1 e s_2:

$x : A : s_1$	$b : B : s_2$	(s_1, s_2)	$\lambda x : A.b$	sistema
$*$	$*$	$(*, *)$	termos que dependem de termos	$\lambda_{\rightarrow}$
$\Box$	$*$	$(\Box, *)$	termos que dependem de tipos	λ2
$*$	$\Box$	$(*, \Box)$	tipos que dependem de termos	λP
$\Box$	$\Box$	$(\Box, \Box)$	tipos que dependem de tipos	$\lambda\underline{\omega}$

7.1.2 O λ-Cubo

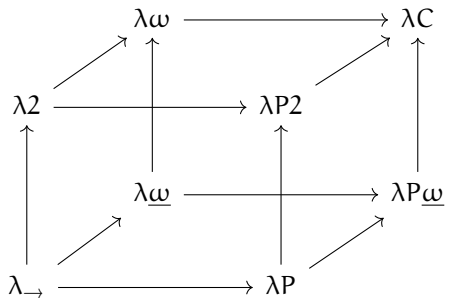
Todas as extensões do cálculo λ simplesmente tipado vistos nos capítulos anteriores se juntam para formar o λC, mas são independentes entre si. Cada sistema diferente pode ser unido ao outro formando outro sistema da seguinte forma:

- $\lambda\underline{\omega} + \lambda 2 = \lambda\omega$
- $\lambda\underline{\omega} + \lambda P = \lambda P\underline{\omega}$
- $\lambda P + \lambda 2 = \lambda P2$

Alguns desses sistemas derivados não possuem nomes separados como as extensões iniciais, sendo vistos primariamente como subsistemas de λC. O primeiro subsistema dessa lista é, só para relembrar, o Sistema $\mathcal{F}\omega$ de Girard.

Barendregt, no seu estulo desses diversos sistemas e suas ligações, desenvolveu uma forma de perceber esses sistemas em uma espécie de Cubo, onde

cada "dimensão" (Largura, altura e profundidade) teria uma característica associada a um sistema diferente da seguinte forma:



As regras de inferência do λC são as seguintes:

(*sort*) $\emptyset \vdash * : \square$

(*var*) $\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$ se $x \notin \Gamma$

(*weak*) $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$ se $x \notin \Gamma$

(*form*) $\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$

(*appl*) $\dfrac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$

(*abst*) $\dfrac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B}{\Gamma \vdash \lambda x : A.M : \Pi x : A.B}$

(*conv*) $\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s}{\Gamma \vdash A : B'}$ se $B =_\beta B'$

7.1.3 Propriedades de λC

A maioria das propriedades de λC são extensões das propriedades vistas anteriormente nos seus subsistemas. O que muda de um para o outro é a necessidade de construir definições e proposições que abarquem as diversas dependências que existem no sistema. Muitas das provas também se tornam mais e mais complexas a medida que são adicionadas novas regras e formas de se relacionar os sorts

Primeiro, é necessário definir as expressões de λC:

Definição 7.1 (Expressões de λC, $\mathcal{E}$, (NEDERPELT; GEUVERS, 2014))**.** O conjunto $\mathcal{E}$ de λC-expressões é definido pela seguinte BNF:

$$\mathcal{E} = V|\square| * |(\mathcal{E}\mathcal{E})|(\lambda V : \mathcal{E}.\mathcal{E})|\Pi V : \mathcal{E}.\mathcal{E}$$

A notação de parenteses, abstrações sucessivas e aplicações sucessivas é a mesma.

Lema 7.1 (*Lema das Variáveis Livres*, (NEDERPELT; GEUVERS, 2014))**.** Se $\Gamma \vdash A : B$, então $FV(A)$ e $FV(B) \subseteq dom(\Gamma)$

O contexto é dito *bem-formado* se ele forma parte de um juizo derivável:

Definição 7.2 ((NEDERPELT; GEUVERS, 2014))**.** Um contexto Γ é *bem formado* se existem A e B tais que $\Gamma \vdash A : B$.

Com isso, é possível ver os seguintes lemas:

Lema 7.2 (Afinamento, Condensação, Permutação, (NEDERPELT; GEUVERS, 2014))**.**

(1) (*Afinamento*) Sejam Γ' e Γ'' contextos tais que $\Gamma' \subseteq \Gamma''$. Se $\Gamma' \vdash A : B$ e Γ'' é bem formado, então $\Gamma'' \vdash A : B$

(2) (*Permutação*) Sejam Γ' e Γ'' dois contextos e seja Γ'' a permutação de Γ'. Se $\Gamma' \vdash A : B$ e Γ'' é bem formado, então $\Gamma'' \vdash A : B$.

(3) (*Condensação*) Se $\Gamma', x : A, \Gamma'' \vdash B : C$ e x não ocorre em Γ'', B ou C, então $\Gamma', \Gamma'' \vdash B : C$.

Também é interessante saber se uma expressão é *legal* ou não:

Definição 7.3 ((NEDERPELT; GEUVERS, 2014))**.** Uma expressão M em λC é *legal* se existe Γ e N tais que $\Gamma \vdash M : N$ ou $\Gamma \vdash N : M$ (Ou seja, quando M é *tipável* ou *habitável*)

Lema 7.3 (Lema da subexpressão, (NEDERPELT; GEUVERS, 2014))**.** Se M é legal, então toda subexpressão de M é legal

Outro lema importante:

Lema 7.4 (Lema da unicidade de tipos a menos que conversão, (NEDERPELT; GEUVERS, 2014))**.** Se $\Gamma \vdash A : B_1$ e $\Gamma \vdash A : B_2$, então $B_1 =_\beta B_2$

Lema 7.5 (Lema da substituição, (NEDERPELT; GEUVERS, 2014))**.** Seja $\Gamma', x : A, \Gamma'' \vdash B : C$ e $\Gamma' \vdash D : A$, então $\Gamma', \Gamma''[x := D] \vdash B[x := D] : C[x := D]$

Esse lema fala que é possível substituir D em $\Gamma', x : A, \Gamma'' \vdash B : C$ caso D e $x : A$ possuam o mesmo tipo. Como tipos também podem possuir termos, essa substituição vai ocorer em todos os lugares.

Um teorema presente também em λC é o Teorema de Church-Rosser:

Teorema 7.1 (Teorema de Church-Rosser, (NEDERPELT; GEUVERS, 2014))**.** A propriedade de Church-Rosser é válida para λC, ou seja, se M está em $\mathcal{E}$, $M \twoheadrightarrow_\beta N_1$ e $M \twoheadrightarrow_\beta N_2$, então existe um N_3 tal que $N_1 \twoheadrightarrow_\beta N_3$ e $N_2 \twoheadrightarrow_\beta N_3$

Corolário 7.1 ((NEDERPELT; GEUVERS, 2014))**.** Suponha que M e N estão em $\mathcal{E}$ e $M =_\beta N$, então existe um L tal que $M \twoheadrightarrow_\beta L$ e $N \twoheadrightarrow_\beta L$

Lema 7.6 (Redução de sujeito, (NEDERPELT; GEUVERS, 2014))**.** Se $\Gamma \vdash A : B$ e $A \twoheadrightarrow_\beta A'$, então $\Gamma \vdash A' : B$

Outro teorema muito importante que está presente em λC é o da normalização forte:

Teorema 7.2 (Teorema da Normalização Forte, (NEDERPELT; GEUVERS, 2014))**.** Todo termo legal M é fortemente normalizável

Finalmente, de um ponto de vista computacional, a questão que fica é saber se a Boa-tipagem e a checagem de tipos é decidível ou não em λC, e a resposta é sim:

Teorema 7.3 (Decidabilidade em λC, (NEDERPELT; GEUVERS, 2014))**.** Em λC e seus subsistemas, as questões de Boa-tipagem e Checagem de tipos são decidíveis

Logo, é possível desenvolver um *programa de computador* que resolve esses problemas automaticamente. Um desses programas é o *Provador de Teoremas* **Coq**.

A questão de Encontrar os termos só é decidível em $\lambda_{\rightarrow}$ e $\lambda\underline{\omega}$, mas indecidível nos outros sistemas.

7.2 A lógica no λC

No capítulo anterior, sobre o sistema λP, foi codificada uma lógica na teoria dos tipos que possuia como operadores a *implicação* e o *quantificador universal*. Como λP é parte de λC, essa codificação pode ser mantida sem problemas.

Para se ter uma lógica proposicional além de *minima*, é necessário introduzir outros conectivos lógicos como a negação ($\neg$), a conjunção ($\wedge$) e a disjunção ($\vee$). Isso não pode ser feito em λP mas pode ser feito em λC

7.2.1 Absurdo e negação na teoria dos tipos

Começando com a negação. É natural de se pensar a negação $\neg A$ como uma implicação $A \Rightarrow \bot$ que leva de A para o "absurdo", chamado de *contradição*. Logo $\neg A$ é lido como "A implica no absurdo".

Para isso, se torna necessário codificar o absurdo, que será codificado na seguinte forma:

$$\bot \equiv \Pi\alpha : *.\alpha$$

. Fica a cargo do leitor mostrar que esse tipo é válido em λC.

Uma característica do absurdo é a chamada *regra da explosão*, ou *ex falso quodlibet* (do falso se segue qualquer coisa):

Se $\bot$ é verdadeiro, então toda proposição é verdadeira

No sentido da teoria dos tipos, é a mesma coisa que dizer que:

Se $\bot$ é habitado, qualquer proposição é habitada

Na interpretação construtiva, pode-se construir uma função:

"Se M é habitante de $\bot$, então existe uma função que mapeia uma proposição arbitrária α em um habitante desse mesmo α"

Essa função possui o tipo $\Pi\alpha : *.\alpha$ e se f possui esse tipo, entao, pela aplicação, $fA : \alpha[\alpha := A] \equiv A$. Ou seja, fA habita A para qualquer tipo A Então:

"Se M for um habitante de $\bot$, então existe uma função f que habita $\Pi\alpha : *.\alpha$"

De forma contrária, tendo f, então todas as proposições são verdadeiras, o que é absurdo, logo existe o absurdo.

Em suma: $\bot$ é habitado, se e somente se, $\Pi\alpha : *.\alpha$ for habitado.

A regra da eliminação de $\bot$ é feita da seguinte forma:

$$(\bot \text{ - } elim)\ \frac{\bot}{A}$$

Na dedução natural, essa regra não aparece com esse nome específico, mas sim relacionado à negação.

Para terminar, é necessário discutir a qual subsistema de $\lambda C \perp$ pertence. No tipo $\Pi\alpha : *.\alpha$, como $\alpha : * : \Box$, $s_1 = \Box$ e $s_2 = *$.

Tendo declarado o absurdo, se torna fácil definir a negação. $\neg A \equiv A \to \perp$ é abreviação de $\Pi x : A.\perp$. Como $A : *$ e $\perp : *$, então $(s_1, s_2) = (*, *)$. Como trata-se de um tipo que usa $\perp$, então seu subsistema mínimo é $\lambda 2$.

É possível definir a regra da introdução do absurdo da seguinte forma:

$$(\perp \text{ - } intro)\ \frac{A \qquad \neg A}{\perp}$$

mas como $\neg A \equiv A \to \perp$, isso é o mesmo que:

$$(\perp \text{ - } intro)\ \frac{A \qquad A \to \perp}{\perp}$$

que é nada mais que uma repaginação da regra ($\Rightarrow$-*elim*). Da mesma forma, as regras de introdução e eliminação da negação também são repaginações das regras de introdução e eliminação da implicação:

$$(\neg \text{ - } elim)\ \frac{A \qquad A \to \perp}{\perp} \qquad (\Rightarrow \text{ - } elim)\ \frac{A \qquad A \Rightarrow B}{B}$$

$$\begin{array}{cc} & [A] \\ & \vdots \\ (\neg\text{-}intro)\ \dfrac{\perp}{\neg A} & (\neg\text{-}intro)\ \dfrac{B}{A \to B} \end{array}$$

7.2.2 Conjunção e Disjunção na teoria dos tipos

I. Conjunção

A conjunção $A \wedge B$ é verdadeira se, e somente se, *ambos* A e B são verdadeiros. Existe uma codificação da conjunção em $\lambda 2$ que é:

$$A \wedge B \equiv \Pi C : *.(A \to B \to C) \to C$$

chamada de *codificação de segunda ordem* da conjunção, sendo mais geral que a codificação *de primeira ordem*:

$$A \wedge B \equiv (A \to_{\neq} B)$$

A codificação de segunda ordem é mais geral pois a codificação de primeira ordem só funciona na lógica clássica.

Mas o que quer dizer de fato essa codificação? Lendo o Π como um quantificador universal, pode-se ler como: "para todo C, (A implica em B implica em C) implica em C". Ou: "Se A e B juntos implicam em C, então C é válido por si só".

Essa codificação é chamada de *segunda ordem* pois ela é generalizada sobre todas as proposições $C : *$ que são objetos de segunda ordem.

As regras de inferência de $\wedge$ na dedução natural e na teoria dos tipos são as seguintes:

$$(\wedge\text{-}intro)\ \frac{A \qquad B}{A \wedge B} \qquad (\wedge\text{-}intro)\ \frac{A \qquad B}{\Pi C : *.(A \to B \to C) \to C}$$

$$(\wedge\text{-}elim\text{-}left)\ \frac{A \wedge B}{A} \qquad (\wedge\text{-}intro\text{-}left)\ \frac{\Pi C : *.(A \to B \to C) \to C}{A}$$

$$(\wedge\text{-}elim\text{-}right)\ \frac{A \wedge B}{B} \qquad (\wedge\text{-}intro\text{-}right)\ \frac{\Pi C : *.(A \to B \to C) \to C}{B}$$

Para provar que essas regras são válidas em λC, é necessário encontrar um termo que obedeça a essas regras da seguinte forma:

$$(\wedge\text{-}intro\text{-}sec\text{-}tt)\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash ?_1 : \Pi C : *.(A \to B \to C) \to C}$$

$$(\wedge\text{-}intro\text{-}sec\text{-}left\text{-}tt)\ \frac{\Gamma \vdash c : \Pi C : *.(A \to B \to C) \to C}{\Gamma \vdash ?_2 : A}$$

$$(\wedge\text{-}intro\text{-}sec\text{-}right\text{-}tt)\ \frac{\Gamma \vdash c : \Pi C : *.(A \to B \to C) \to C}{\Gamma \vdash ?_3 : B}$$

Como exemplo, segue a derivação da regra de introdução:

$$? \ \frac{\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\cdots}}{\Gamma \vdash ?_1 : \Pi C : *.(A \to B \to C) \to C}$$

$$(appl)\ \frac{\Gamma, C : *, z : (A \to B \to C) \vdash b : B \qquad (appl)\ \dfrac{\Gamma, C : *, z : (A \to B \to C) \vdash a : A \qquad \Gamma, C : *, z : (A \to B \to}{\Gamma, C : *, z : (A \to B \to C) \vdash za : B \to C}}{(abst)\ \dfrac{\Gamma, C : *, z : (A \to B \to C) \vdash zab : C}{(abst)\ \dfrac{\Gamma, C : * \vdash \lambda z : (A \to B \to C).zab : (A \to B \to C) \to C}{\Gamma \vdash \lambda C : *.\lambda z : (A \to B \to C).zab : \Pi C : *.(A \to B \to C) \to C}}}$$

Com $\Gamma \equiv A : *, B : *, a : A, b : B$

II. Disjunção

Para a disjunção também existe uma codificação de primeira e segunda ordem. A codificação de segunda ordem é:

$$A \vee B \equiv \Pi C : *.(A \to C) \to (B \to C) \to C$$

(A codificação de primeira ordem é $A \vee B \equiv \neg A \to B$, mas assim como na conjunção, ela só serve para a lógica clássica)

Assim como na conjunção, essa codificação possui uma interpretação literal como:

Para todo C, *(*A → C *implica que (*B → C *implica em* C*))*

Isso é o mesmo que dizer que:

"Se A implica em C e B implica em C, então C se segue"

Ou

"Se A ou B implica em em C, então C se segue"

As regras de inferência de ∨ na dedução natural e na teoria dos tipos são as seguintes:

$$(\vee\text{-}intro\text{-}left)\ \frac{A}{A \vee B} \qquad (\vee\text{-}intro\text{-}left)\ \frac{A}{\Pi C : *.(A \to C) \to (B \to C) \to C}$$

$$(\vee\text{-}intro\text{-}right)\ \frac{B}{A \vee B} \qquad (\vee\text{-}intro\text{-}right)\ \frac{B}{\Pi C : *.(A \to C) \to (B \to C) \to C}$$

$$(\vee\text{-}elim)\ \frac{A \vee B \qquad A \Rightarrow C \qquad B \Rightarrow C}{B}$$

$$(\vee\text{-}elim)\ \frac{\Pi D : *.(A \to D) \to (B \to D) \to D \qquad A \to C \qquad B \to C}{C}$$

Para as regras da dedução natural:

As regras (*intro*) são o mesmo que: se A é verdadeira por si só, então $A \vee B$ também será. O mesmo para B.

A regra (*elim*) descreve bem o comportamento dito acima da implicação

A derivação da (*elim*) pode ser feita seguindo o seguinte termo:

$$\lambda x : \Pi D : *.(A \to D) \to (B \to D) \to D.\lambda y : A \to C.\lambda z : B \to C.xCyz : C$$

A árvore de derivação fica para ser construida pelo leitor.

Uma nota importante é que os operadores lógicos vistos até então podem ser construidos quantificando em cima de qualquer proposição(-como-tipo), nesse caso eles se tornam:

$$\begin{aligned}
\neg &\equiv \lambda\alpha : *.(\alpha \to \bot) \\
\wedge &\equiv \lambda\alpha : *.\lambda\beta : *.\Pi\gamma : *(\alpha \to \beta \to \gamma) \to \gamma \\
\vee &\equiv \lambda\alpha : *.\lambda\beta : *.\Pi\gamma : *(\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma
\end{aligned}$$

Exemplo:

$$\begin{aligned}
A \wedge B &\equiv (\lambda\alpha : *.\lambda\beta : *.\Pi\gamma : *(\alpha \to \beta \to \gamma) \to \gamma)AB \\
&\to_\beta (\lambda\beta : *.\Pi\gamma : *(A \to \beta \to \gamma) \to \gamma)B \\
&\to_\beta \Pi\gamma : *(A \to B \to \gamma) \to \gamma \\
&\equiv \Pi C : *(A \to B \to C) \to C
\end{aligned}$$

Enquanto os outros operadores menos gerais estão em $\lambda 2$, esses mais gerais estão no sistema $\lambda\omega$

III. Biimplicação

A Biimplicação, ou também "Se, e somente se", na lógica é o operador $\Leftrightarrow$ que não possui uma tipagem direta em λC, mas pode ser construido usando a seguinte equivalência:

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$$

7.2.3 Exemplo de Derivação

Para mostrar que a lógica funciona na teoria dos tipos, é importante demonstrar um exemplo de tautologia, como a seguinte:

$$(A \vee B) \Rightarrow (\neg A \Rightarrow B)$$

Como cada operador desse possui uma codificação na teoria dos tipos, essa tautologia pode ser reescrita na forma:

$$(\Pi C : *.(A \to C) \to (B \to C) \to C) \to (A \to \bot) \to B$$

Como A e B são proposições arbitrárias, elas ficam no contexto da derivação.

Para dar um norte, a derivação na dedução natural dessa tautologia é a seguinte:

$$\dfrac{\dfrac{\dfrac{}{(\vee\textit{-elim})}\,[A \vee B]^1 \qquad \dfrac{\dfrac{[A]^3 \qquad [\neg A]^2}{(\neg\textit{-elim})}}{(\bot\textit{-elim})\ \dfrac{\bot}{B}} \qquad [B]^3}{(\Rightarrow\textit{-intro})\ \dfrac{B}{\neg A \Rightarrow B}\ 2}\ 3}{(\Rightarrow\textit{-intro})\ \dfrac{}{(A \vee B) \Rightarrow (\neg A \Rightarrow B)}\ 1}$$

Essa árvore pode dar uma noção de por onde começar

Primeiro, é necessário um termo $c : (\Pi C : *.(A \to C) \to (B \to C) \to C)$ para ser o $A \vee B$. Também de um termo $n : A \to \bot$ para ser o $\neg A$.

Mas a dúvida então fica em como encontrar o B. para isso fica claro pela dedução natural que o próximo passo seria eliminar o $\vee$ usando B no lugar de C, logo seria necessário o tipo: $(A \to B) \to (B \to B) \to B$, gerado por cB. Dado esse tipo, só se torna necessário encontrar dois termos com tipos $A \to B$ e $B \to B$. O segundo é um termo identidade $\lambda v : B.v$ e o primeiro tem que ser um termo $\lambda u : A.M : A \to B$.

Para gerar o termo M é necessário lembrar que o tipo absurdo $\bot$ é o mesmo que $\Pi\alpha : *.\alpha$ e que $(\Pi\alpha : *.\alpha)B \to_\beta B$. Logo, $M \equiv uvB$.

Fica-se então com o seguinte termo:

$$\begin{aligned}\lambda c : (\Pi C : *.(A \to C) \to (B \to C) \to C).\lambda n : (A \to \bot).cB(\lambda u : A.nuB)(\lambda v : B.v)\\ : (\Pi C : *.(A \to C) \to (B \to C) \to C) \to (A \to \bot) \to B\end{aligned}$$

A árvore de derivação completa (retirando a segunda premissa da abstração e fins que não são fins de verdade) está na próxima página:

$$
\dfrac{\dfrac{\dfrac{\Gamma \vdash c : \Pi C : *.(A \to C) \to (B \to C) \to C \qquad \Gamma \vdash B : *}{\Gamma \vdash cB : (A \to B) \to (B \to B) \to B}(appl) \qquad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma, u : A \vdash n : A \to \bot \qquad \Gamma, u : A \vdash u : A}{\Gamma, u : A \vdash nu : \bot \equiv \Pi\alpha : *.\alpha}(appl) \qquad \Gamma, u : A \vdash B : *}{\Gamma, u : A \vdash nuB : B}(appl)}{\Gamma \vdash \lambda u : A.nuB : A \to B}(abst) \qquad \dfrac{\Gamma, v : B \vdash v : B}{\Gamma \vdash \lambda v : B.v : (B \to B)}(abst)}{\Gamma \vdash cB(\lambda u : A.nuB) : (B \to B) \to B}(appl)}{A : *, B : *, c : (\Pi C : *.(A \to C) \to (B \to C) \to C), n(A \to \bot) \vdash cB(\lambda u : A.nuB)(\lambda v : B.v) : B}(abst)}{A : *, B : *, c : (\Pi C : *.(A \to C) \to (B \to C) \to C) \vdash \lambda n : (A \to \bot)..cB(\lambda u : A.nuB)(\lambda v : B.v) : (A \to \bot) \to B}(abst)}{A : *, B : * \vdash \lambda c : (\Pi C : *.(A \to C) \to (B \to C) \to C).\lambda n : (A \to \bot).cB(\lambda u : A.nuB)(\lambda v : B.v) : (\Pi C : *.(A \to C) \to (B \to C) \to C) \to (A \to \bot) \to B}
$$

7.2.4 Lógica clássica em λC

A lógica vista até então não é *clássica*, mas sim *intuicionista* ou *construtivista*. Essa lógica é um pouco mais fraca que a lógica clássica, pois ela não possui a *lei do terceiro excluido* (LEM) que diz que $A \vee \neg A$ é válido para qualquer A. Ela também não possui a lei da *dupla negação* (DN) que diz que $\neq\neq A \Rightarrow A$ é válido para qualquer A.

Para obter essa lógica clássica é necessário fazer uma extensão da lógica vista até então. Para isso, não é necessário introduzir ambas LEM e DN, pois uma implica na outra, dessa forma só será introduzida a LEM.

Mas como introduzir uma regra que não é derivável em λC? para isso, é necessário assumi-la como *suposição*, também chamado de *postulado* ou *axioma*, no contexto. A LEM será introduzida da seguinte forma:

$$i_{LEM} = \Pi\alpha : *.\alpha \vee \neg\alpha$$

Um teste inicial interessante é provar o que foi dito anteriormente que $\lambda C + i_{LEM}$ prova *DN*.

$$? \frac{?}{i_{LEM} \vdash ? : \Pi\beta : *.\neg\neg\beta \to \beta}$$

$$(abst)\ \frac{? \dfrac{?}{i_{LEM}, \beta : * \vdash ? : \neg\neg\beta \to \beta}}{i_{LEM} \vdash \lambda\beta : *.? : \Pi\beta : *.\neg\neg\beta \to \beta}$$

$$(abst)\ \frac{(abst)\ \dfrac{? \dfrac{?}{i_{LEM}, \beta : *, x : \neg\neg\beta \vdash ? : \beta}}{i_{LEM}, \beta : * \vdash \lambda x : \neg\neg\beta.? : \neg\neg\beta \to \beta}}{i_{LEM} \vdash \lambda\beta : *.\lambda x : \neg\neg\beta.? : \Pi\beta : *.\neg\neg\beta \to \beta}$$

Nessa etapa, é necessário descobrir um termo $M : \beta$. Mas tendo em vista que $i_{LEM}\beta \equiv \beta \vee \neg\beta \equiv \Pi\gamma : *.(\beta \to \gamma) \to (\neg\beta \to \gamma) \to \gamma$, é possível construir o tipo $i_{LEM}\beta \equiv \beta \equiv (\beta \to \beta) \to (\neg\beta \to \beta) \to \beta$.

A primeira entrada desse tipo pode ser construida usando um termo que é uma identidade $\lambda u : \beta.u$, já o segundo pode ser construido pensando que $\neg\neg\beta \equiv (\neg\beta \to \bot)$. Sendo $x : \neg\neg\beta$, ao aplicar esse termo a um termo $z : \neg\beta$, tem-se $xz : \bot$, mas pela definição de $\bot$, $\bot\beta \to_\beta \beta$. logo $xz\beta : \beta$. Então o segundo termo é: $\lambda z : \neg\beta.xz\beta : \neg\beta \to \beta$.

O termo M então se torna $M \equiv i_{LEM}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta$.

A árvore final se torna:

Obs: árvore incompleta; $\Gamma \equiv i_{LEM}, \beta : *, x : \neg\neg\beta$

$$
\dfrac{
\dfrac{
\dfrac{\vdots}{\Gamma \vdash i_{\mathrm{LEM}}\beta\beta : (\beta \to \beta) \to (\neg\beta \to \beta) \to \beta}\,(\mathit{appl})
\qquad
\dfrac{\Gamma, y : \beta \vdash y : \beta \to \beta}{\Gamma \vdash \lambda y : \beta.y : \beta \to \beta}\,(\mathit{abst})
}{\Gamma \vdash i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y) : (\neg\beta \to \beta) \to \beta}\,(\mathit{appl})
\qquad
\dfrac{
\dfrac{
\dfrac{\Gamma, z : \neg\beta \vdash x : \neg\neg\beta \qquad \Gamma, z : \neg\beta \vdash z : \neg\beta}{\Gamma, z : \neg\beta \vdash xz : \bot \equiv \Pi\alpha : *.\alpha}\,(\mathit{appl})
\qquad
\Gamma, z : \neg\beta \vdash \beta : *
}{\Gamma, z : \neg\beta \vdash xz\beta : \beta}\,(\mathit{appl})
}{\Gamma \vdash \lambda z : \neg\beta.xz\beta : \beta}\,(\mathit{abst})
}{i_{\mathrm{LEM}}, \beta : *, x : \neg\neg\beta \vdash i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta}\,(\mathit{appl})
$$

$$
(\mathit{appl})\ \dfrac{i_{\mathrm{LEM}}, \beta : *, x : \neg\neg\beta \vdash i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta}{i_{\mathrm{LEM}}, \beta : *, x : \neg\neg\beta \vdash i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \beta}
$$

$$
(\mathit{abst})\ \dfrac{}{i_{\mathrm{LEM}}, \beta : * \vdash \lambda x : \neg\neg\beta.i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \neg\neg\beta \to \beta}
$$

$$
(\mathit{abst})\ \dfrac{}{i_{\mathrm{LEM}} \vdash \lambda\beta : *.\lambda x : \neg\neg\beta.i_{\mathrm{LEM}}\beta\beta(\lambda y : \beta.y)(\lambda z : \neg\beta.xz\beta) : \Pi\beta : *.\neg\neg\beta \to \beta}
$$

7.2.5 Lógica de predicados em λC

Uma vez tendo definido as codificações relacionadas a lógica proposicional, se torna necessário codificar a *lógica de predicados*. A lógica de predicados pode ser vista como uma extensão da lógica proposicional com o quantificador universal ($\forall$) e o quantificador existencial ($\exists$). Na discussão de λP, foi construida a codificação de $\forall_{x \in S} P(x)$ como $\Pi x : S.Px$. O que falta seria a definição do quantificador existencial. Uma definição de *primeira ordem* de $\exists$ seria simplesmente $\exists_{x \in S} P(x) \equiv \neg \forall_{x \in S} P(x)$, mas essa codificação só funciona na lógica clássica. Uma codificação mais geral seria a codificação de segunda ordem de $\exists$ na forma:

$$\exists_{x \in S} P(x) \equiv \Pi \alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$$

Traduzindo em palavras:

"Para todo α: se é sabido que para todo x em S se segue que Px implica em α, então α"

Não é imediatamente clara a relação entre essa codificação é o sentido do quantificador existencial, mas ela obecede as regras de inferencia de $\exists$.

A regra da eliminação do existencial diz que:

$$\exists\text{-}elim \ \frac{\exists_{x \in S} P(x) \qquad \forall_{x \in S}(P(x) \Rightarrow A)}{A}$$

Essa regra é:

- primeira premissa: se existe um x no conjunto S para o qual o predicado P é válido,
- segunda premissa: e para todo x em S, se P é válido para x então A é válida
- conclusão: então A é válido

A contraparte dessa regra na codificação mostrada anteriormente é:

$$\exists\text{-}elim\text{-}sec \ \frac{\Pi \alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \qquad \Pi x : S.(Px \to A)}{A}$$

O termo que prova essa regra é o seguinte:

$$\lambda y : \Pi \alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha).\lambda z : \Pi x : S.(Px \to A).yAz$$

Abreviando $\Pi x : S.(Px \to \alpha)$ para $\phi(\alpha)$, então a regra expressa da seguinte forma:

"Se para todo α tem-se que $\phi(\alpha) \Rightarrow \alpha$ e $\phi(A)$, então A"

Para que essa derivação seja correta, é necessário que $x \notin |FV(A)$, pois pelo contrario a aplicação yA seria ilegal.

Já para a introdução de $\exists$ a regra de derivação é a seguinte:

$$\exists\text{-}intro \ \frac{a \in S \qquad P(a)}{\exists_{x \in S} P(x)}$$

sendo a algum elemento fixo.
A regra da introdução na teoria dos tipos fica:

$$\exists\text{-}intro\text{-}sec\ \frac{a : S \qquad Pa}{\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)}$$

A árvore de derivação dessa regra é a seguinte:

$$(appl)\ \frac{S : *, P : S \to *, a : S, u : Pa, \alpha : *, v : (\Pi x : S.(Px \to \alpha)) \vdash v : (\Pi x : S.(Px \to \alpha)) \qquad S : *, P : S \to *, a : S}{(appl)\ \dfrac{S : *, P : S \to *, a : S, u : Pa, \alpha : *, v : (\Pi x : S.(Px \to \alpha)) \vdash va : (Pa \to}{(abst)\ \dfrac{S : *, P : S}{(abst)\ \dfrac{S : *, P : S}{S : *, P : S \to *, a : S, u : Pa}}}}$$

Uma representação alternativa do existencial é o seguinte:

$$\exists \equiv \lambda S : *.\lambda P : S \to *.\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha)$$

Nesse caso, $\exists SP$ é a codificação na teoria dos tipos de $\exists_{x \in S} P(x)$

7.2.6 Exemplo de lógica de predicado em λC

A título de exemplo, será feita a derivação da seguinte tautologia:

$$\neg\exists_{x \in S} P(x) \Rightarrow \forall_{y \in S} \neg P(y)$$

Na codificação de segunda ordem em λC isso se torna:

$$(\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot) \to \Pi y : S.(Py \to \bot)$$

Sendo $\Gamma \equiv S : *, P : S \to *$, o problema se torna:

$$?\ \frac{?}{\Gamma \vdash ? : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot) \to \Pi y : S.(Py \to \bot)}$$

Usando abstração nos primeiros passos:

$$(abst)\ \frac{(abst)\ \dfrac{(abst)\ \dfrac{?}{\Gamma, u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot), y : S, v : Py \vdash ? : \bot}}{\Gamma, u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot), y : S \vdash \lambda v : Py.? : Py \to \bot}}{\Gamma, u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot) \vdash \lambda y : S.\lambda v : Py.? : \Pi y : S.(Py \to \bot)}$$
$$(abst)\ \frac{}{\Gamma \vdash \lambda u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot).\lambda y : S.\lambda v : Py.? : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot}$$

Agora a questão se torna querer encontrar esse termo que gere o absurdo $\bot$.
Para isso, é necessário usar a negação em u para gerar um absurdo. É necessário gerar um termo de tipo $\exists_{x \in S} P(x)$, como o seguinte:

$$\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv : \exists_{x \in S} P(x)$$

Tendo gerado esse tipo, é possível ver que a sua aplicação com u gera o absurdo:

$$u(\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv) : \bot$$

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash u : \exists_{x\in S} P(x) \to \bot \qquad \Gamma' \vdash \lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv : \exists_{x\in S} P(x)}{\Gamma' \vdash u(\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv) : \bot}\,(appl)}{\Gamma, u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot), y : S \vdash \lambda v : Py.u(\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv) : Py \to \bot}\,(abst)}{\Gamma, u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot) \vdash \lambda y : S.\lambda v : Py.u(\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv) : \Pi y : S.(Py \to \bot)}\,(abst)}{\Gamma \vdash \lambda u : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot).\lambda y : S.\lambda v : Py.u(\lambda\alpha : *.\lambda w : \Pi x : S.(Px \to \alpha).wyv) : (\Pi\alpha : *.((\Pi x : S.(Px \to \alpha)) \to \alpha) \to \bot) \to \Pi y : S.(Py \to \bot)}\,(abst)
$$

Part II
Construções paralelas ao cubo

Part III
Semântica Categorial das teorias do cubo lambda

8 Introdução à Teoria das Categorias

A teoria das categorias é uma área de matemática que relaciona diversas áreas, como por exemplo, Teoria dos Grupos, Teoria dos Anéis, Topologia, Teoria dos Grafos, etc. Cada uma dessas teorias tem em comum a definição de seus objetos (Grupos, Anéis, Espaços topológicos, grafos) e formas de relacionar esses objetos (Homomorfismos de grupos, homomorfismos de aneis, homeomorfismos, homomorfismos entre grafos).

8.1 Categorias

Para estudar categorias, primeiro é necessário defini-las:

Definição 8.1 (Categoria, (AWODEY, 2010))**.** Uma *categoria* **C** consiste em:

- *Objetos*: $A, B, C, \ldots$
- *Setas* (Morfismos): $f, g, h, \ldots$
- Para cada seta f existem objetos:

  $$dom(f), cod(f)$$

  chamados de *domínio* e *contradomínio* de f. A escrita

  $$f : A \to B$$

  indica que $A = dom(f)$ e $B = cod(f)$
- Sejam setas $f : A \to B$ e $g : B \to C$ com:

  $$cod(f) = dom(g)$$

  existe uma seta $g \circ f : A \to C$ chamada de *composição* de f com g
- Para cada objeto A existe uma seta

  $$1_A : A \to A$$

  chamada de *seta identidade* de A

Esses dados precisam satisfazer os seguintes axiomas:

- (Associatividade) Sejam $f : A \to B$, $g : B \to C$ e $h : C \to D$ setas, então:

  $$h \circ (g \circ f) = (h \circ g) \circ f$$

- (Identidade) Seja $f : A \to B$ uma seta, então

$$f \circ 1_A = f = 1_B \circ f$$

Para quaisquer objetos A e B em uma categoria C, a coleção de setas de A para B é escrito $\mathrm{Hom}_C(A, B)$

Alguns exemplos de categorias são:

1. A categoria **Set** que possui conjuntos como objetos e funções como morfismos.

2. Os conjuntos ordenados descritos na Definição 1.31 também podem formar uma categoria junto com os mapeamentos monótonos descritos na Definição 1.32, chamada de **Pos**

3. Um monóide é um conjunto M equipado com uma operação binária $\cdot : M \times M \to M$ e um elemento unitário $e \in M$ tal que para todo $x, y, z \in M$:

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

e

$$e \cdot x = x = x \cdot e$$

. Por exemplo, o conjunto dos naturais $\mathbb{N}$, junto à operação de soma usual $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, pode ser considerado um monoide, com o 0 como elemento unitário.
Dois monóides $(M, \cdot)$ e $(N, \star)$ podem ser relacionados através de um *homomorfismo* $\phi : M \to N$ tal que

$$\phi(x \cdot y) = \phi(x) \star \phi(y)$$

e

$$\phi(e_M) = e_N$$

A categoria que possui monóides como objetos e homeomorfismos como morfismos é denominada de **Mon**

4. Um grupo G é um monóide onde para todo $a \in G$ existe um elemento $b \in G$ tal que $a \cdot b = e$. b é chamado de *inverso* de a e é escrito como a^{-1}. Um homomorfismo ϕ entre dois grupos $(G, \cdot)$ e $(H, \star)$ obedece as duas condições para homomorfismos entre monóides mais a seguinte:

$$\phi(a^{-1}) = \phi(a)^{-1}$$

A categoria que possui monóides como objetos e homomorfismos como morfismos é denominada de **Grp**

5. ((RIEHL, 2017)) Um grupo G (e também um monóide) define uma categoria BG com um único objeto. Os elementos do grupo são seus morfismos e a composição é dada por $\cdot$. O elemento unitário $e \in G$ age como o morfismo identidade para o objeto único dessa categoria.
Por exemplo, para $(\mathbb{Z}, +)$, $e = 0$ e será representado por $0 : \mathbb{Z} \to \mathbb{Z}$. Sendo $1 : \mathbb{Z} \to \mathbb{Z}$ e $2 : \mathbb{Z} \to \mathbb{Z}$, então a composição $1 \circ 2$ é em $(\mathbb{Z}, +)$ equivalente a $1 + 2$ e $1 \circ 2 = 3$.

Definição 8.2 (Isomorfismos, (AWODEY, 2010))**.** Em qualquer categoria C, um morfismo $f : A \to B$ é chamado de *isomorfismo* se existe um morfismo $g : B \to A$ em C tal que

$$g \circ f = 1_A \text{ e } f \circ g = 1_B$$

g é chamado de inverso de f e, por ser único, pode ser denotado por f^{-1}. Os objetos A e B são ditos *isomórficos* e denotados por $A \cong B$

Exemplos:

1. Os isomorfismos em **Set** são bijeções
2. Os isomorfismos em **Grp** são os homomorfismos bijetivos

Definição 8.3 (Categorias pequenas, (AWODEY, 2010))**.** Uma categoria C é chamada de *pequena* se a coleção C_0 de objetos em C e a coleção C_1 de morfismos em C são conjuntos. Caso contrário, C é chamada de *grande*

Todas as categorias finitas são pequenas, assim como a categoria $\mathtt{Sets}_{\mathtt{fin}}$ de conjuntos finitos. Já a categoria **Sets** é grande (Pois caso a coleção de seus objetos fosse um conjunto, isso geraria o paradoxo de Russell)

Definição 8.4 (Categoria localmente pequena, (AWODEY, 2010))**.** Uma categoria C é chamada de *localmente pequena* se para quaisquer objetos X e Y em C, a coleção de morfismos $\mathtt{Hom}_C(X, Y) = \{f \in C_1 | f : X \to Y\}$ é um *conjunto* (Chamado de *hom-set*)

8.2 Categorias novas das antigas

Dada a definição de categorias, é interessante analisar o que pode ser feito com uma categoria e como gerar novas categorias de categorias antigas

Definição 8.5 (Categoria oposta, (AWODEY, 2010))**.** A categoria *oposta* (ou "dual") C^{op} de uma categoria C possui os mesmos objetos que C, mas para cada morfismos $f : A \to B$ em C existe um morfismo $f : B \to A$ em C^{op}

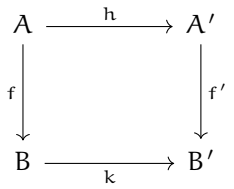
A categoria oposta inverte todos os morfismos da categoria que parte. Então seja f^{op} o morfismo invertido, a composição na categoria oposta se torna: $f^{op} \circ g^{op} = (g \circ f)^{op}$

É interessante perceber que cada resultado na Teoria das Categorias terá um resultado dual ganho "de graça" ao fazer esse resultado nas categorias duais.

Também é possível ver que $(C^{op})^{op} = C$

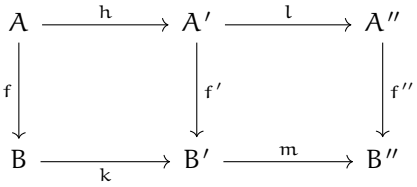
Definição 8.6 (Categoria de setas, (ROSIAK, 2022))**.** Seja uma categoria C, definimos a *categoria de setas* de C, denotada por $C^{\to}$, tendo:

- Objetos: morfismos $A \xrightarrow{f} B$ de C
- Morfismos: a partir de um objeto de $C^{\to}$ $A \xrightarrow{f} B$ para outro $A' \xrightarrow{f'} B'$ um morfismo é um par $\langle A \xrightarrow{f} B, A' \xrightarrow{f'} B' \rangle$ de morfismos de C fazendo o diagrama



comutar. Ou seja, $k \circ f = f' \circ h$ em C

A composição das setas é feita ao colocar quadrados comutativos lado a lado da seguinte forma:

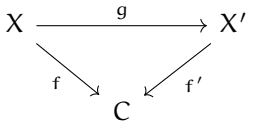


tal que $\langle l, m\rangle \circ \langle h, k\rangle = \langle l \circ h, m \circ k\rangle$
A identidade de um objeto $A \xrightarrow{f} B$ é dado pelo par $\langle id_A, id_B\rangle$

Outro tipo de categoria de interesse é a categoria slice:

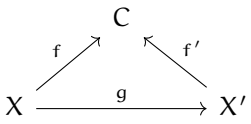
Definição 8.7 (Categoria Slice, (AWODEY, 2010))**.** A categoria slice $\mathbf{C}/C$ de uma categoria **C** sobre um objeto $C \in \mathbf{C}$ possui:

- Objetos: todas as setas $f \in \mathbf{C}$ tal que $cod(f) = C$
- Morfismos: g de $f : X \to C$ e $f' : X' \to C$ é uma seta $g : X \to X'$ em **C** tal que $f' \circ g = f$ como no diagrama:



A composição desses morfismos é basicamente a junção de desses triangulos

Também é possível definir a categoria $(C/\mathbf{C})$ chamada de categoria de co-slice, onde os objetos são setas f de **C** tal que $dom(f) = C$ e uma seta entre $f : C \to X$ e $f' : C \to X'$ é uma seta $h : X \to X'$ tal que $h \circ f = f'$ como no diagrama:



Também é possível definir a noção de subcategoria:

Definição 8.8 (Subcategoria, (ROSIAK, 2022))**.** Uma categoria **D** dita *subcategoria* de **C** é obtida restringindo a coleção de objetos de **C** para uma subcoleção (Ou seja, todo **D**-objeto é um **C**-objeto) e a coleção de morfismos é obtida restringindo a coleção de morfismos de **C** onde:

- Se o morfismo $f : A \to B$ está em **D**, então A e B estão em **D**
- Se A está em **D**, então também está o morfismo identidade id_A
- Se $f : A \to B$ e $g : B \to C$ estão em **D**, então $g \circ f : A \to C$ também está

e também:

Definição 8.9 (Subcategoria cheia, (ROSIAK, 2022))**.** Seja **D** uma subcategoria de **C**. ENtão **D** é uma *subcategoria cheia* de **C** quando **C** não possui setas $A \to B$ além dos que já existem em **D**. Ou seja para quaisquer objetos A e B em **D**, **C**:

$$\mathrm{Hom}_{\mathbf{D}}(A, B) = \mathrm{Hom}_{\mathbf{C}}(A, B)$$

Exemplo:

- A categoria **FinSet** de conjuntos finitos é uma subcategoria de **Set**.
- Um grupo $(G, \cdot)$ é dito *abeliano*, ou comutativo, caso para quaisquer dois elementos $a, b \in G$, $a \cdot b = b \cdot a$. A categoria de grupos abelianos **Ab** é uma subcategoria (cheia) de **Grp**

Um produto de dois conjuntos A e B é dado por

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

. O conjunto produto possui duas projeções:

$$A \xleftarrow[\pi_1]{} A \times B \xrightarrow[\pi_2]{} B$$

tais que

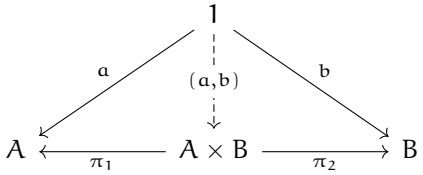
$$\pi_1(a, b) = a, \quad \pi_2(a, b) = b$$

.

Dado um elemento $c \in A \times B$, tem-se que

$$c = (\pi_1 c, \pi_2 c)$$

capturado no seguinte diagrama:



Para categorias gerais:

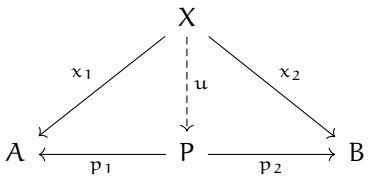
Definição 8.10 ((AWODEY, 2010))**.** Em uma categoria $\mathcal{C}$, um *diagrama de produto* para objetos A e B consiste de um objeto P e setas:

$$A \xleftarrow[p_1]{} P \xrightarrow[p_2]{} B$$

Tais que para qualquer diagrama:

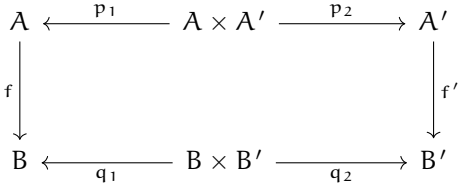
$$A \xleftarrow[x_1]{} X \xrightarrow[x_2]{} B$$

existe uma seta única $u : X \to P$ fazendo o diagrama



comutar, ou seja $x_1 = p_1 u$ e $x_2 = p_2 u$

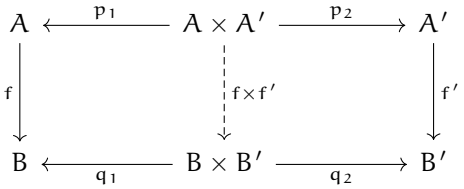
Seja $\mathcal{C}$ uma categoria com diagramas de produto para cada par de objetos, tem-se então:



Então, pode-se encontrar um morfismo

$$f \times f' : A \times A' \to B \times B'$$

para $f \times f' = \langle f \circ p_1, f' \circ p_2 \rangle$ que faz os dois lados do seguinte diagrama comutarem:



Uma categoria que possui um produto para cada par de objetos é dita *ter produtos binários*. Essa construção também pode ser feita para produtos ternários e generalizada para produtos I-ários

8.3 Funtores

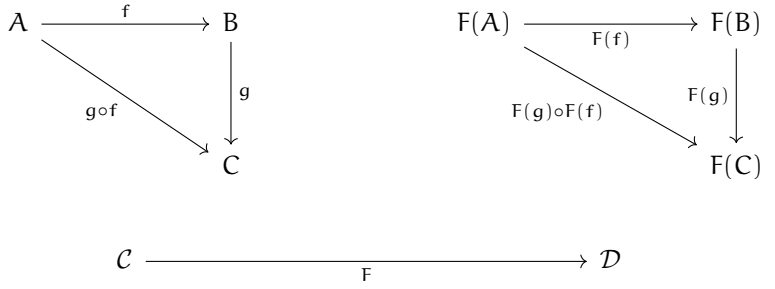
Sendo categorias estruturas que se iniciam com objetos e morfismos entre esses objetos, é natural se perguntar se existem morfismos entre categorias. Esses morfismos deveriam também manter a estrutura entre categorias, como por exemplo os morfismos entre grupos mantém a estrutura do grupo, mesmo que mudando-se os objetos e os morfismos internos à categoria. Esse tipo de morfismo entre categorias é chamado de *funtor* e definido da seguinte forma:

Definição 8.11 (Funtor, (AWODEY, 2010))**.** Um funtor $F : \mathcal{C} \to \mathcal{D}$ entre categorias $\mathcal{C}$ e $\mathcal{D}$ é um mapeamento de objetos em objetos e setas em setas tal que:

1. $F(f : A \to B) = F(f) : F(A) \to F(B)$
2. $F(g \circ f) = F(g) \circ F(f)$

3. $F(1_A) = 1_{F(A)}$

A primeira parte define que para cada objeto A em $\mathcal{C}$, existe um objeto correspondente $F(A)$ em $\mathcal{D}$. Também define que Funtores preservam os domínios e codomínios de cada morfismo.



A segunda parte define que existindo uma composição de morfismos em $\mathcal{C}$, também existira uma composição correspondente em $mathcalD$.

A terceira parte define que as identidades também são preservadas.

Em algumas partes da matemática porém, os funtores não preservam a ordem dos morfismos. Os funtores que preservam como da regra 1 da parte anterior são chamados de *funtores covariantes*. É interessante também definir os funtores *contravariantes*:

Definição 8.12 (Funtor Contravariante, (AWODEY, 2010))**.** Um funtor da forma $F : \mathcal{C}^{op} \to \mathcal{D}$ é chamado de *funtor contravarinate* em $\mathcal{C}$. Ou seja, as regras se tornam:

1. Seja $f : A \to B$ em $\mathcal{C}$, então: $F(f : A \to B) : F(B) \to F(A)$
2. Seja $g \circ f$ em $\mathcal{C}$, então $F(g \circ f) = F(f) \circ F(g)$
3. $F(1_A) = 1_{F(A)}$

Um exemplo essencial de funtor contravariante são os *pré-feixes*, definidos da seguinte forma:

Definição 8.13 (Pré-feixe, (ROSIAK, 2022))**.** Um *pré-feixe* (com valor de conjunto) em $\mathcal{C}$, onde $\mathcal{C}$ é uma categoria pequena, é um funtor $\mathcal{C}^{op} \to$ **Set**

O pré-feixe pode ser visto como uma atribuição de dados locais de acordo com a estrutura de $\mathcal{C}$.

Uma vez definidos funtores, o próximo passo é se perguntar se existe uma categoria que usa funtores como morfismos entre seus objetos, e a resposta é que existe tal categoria, onde objetos são categorias, definida da seguinte forma:

Definição 8.14 (**Cat**, (ROSIAK, 2022))**.** A categoria de *categorias pequenas*, denotada por **Cat**, é a categoria que possui:

- objetos: categorias pequenas
- morfismos: funtores entre elas

Para demonstrar que essa é de fato uma categoria, é necessário definir um morfismo/funtor identidade e a ideia de composição entre morfismos/funtores.

Definição 8.15 ((ROSIAK, 2022))**.** Dada uma categoria $\mathcal{C}$, o *funtor identidade* é o funtor $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ que faz o esperado: leva um objeto a ele mesmo e um morfismos a ele mesmo tal que:

- $\mathrm{id}_{\mathcal{C}}(c) = c$
- $\mathrm{id}_{\mathcal{C}}(f) = f$

Para a composição, sejam $\mathcal{C}$, $\mathcal{D}$ e $\mathcal{E}$ categorias pequenas, e $F : \mathcal{C} \to \mathcal{D}$ e $G : \mathcal{D} \to \mathcal{E}$ dois funtores, a composição de G com F, o funtor composição $G \circ F : \mathcal{C} \to \mathcal{E}$, é definido tal que para objetos c em $\mathcal{C}$, $(G \circ F)(c) = G(F(c))$ e para um morfismo $f : c \to c'$ em $\mathcal{C}$ $(G \circ F)(f) = G(F(f))$. Para definir que $G \circ F$ é um funtor é necessário pedir sua *funtorialidade*, ou seja, que ele obedeça às regras impostas na definição de funtores:
(1) Sejam f, g funtores em $\mathcal{C}$ tal que $g \circ f$ também está definido em $\mathcal{C}$, então:

$$\begin{aligned}(G \circ F)(g \circ f) &= G(F(g \circ f)) \\ &= G(F(g) \circ F(f)) \\ &= G(F(g)) \circ G(F(f)) \\ &= (G \circ F)(g) \circ (G \circ F)(f)\end{aligned}$$

onde a primeira e a última linha são a definição da composição de funtores e o méio é a derivação dessa composição.
(2) Seja c qualquer objeto em $\mathcal{C}$, então:

$$\begin{aligned}(G \circ F)(1_c) &= G(F(1_C)) \\ &= G(1_{F(c)}) \\ &= 1_{G(F(c))} \\ &= 1_{(G \circ F)(c)}\end{aligned}$$

A seguir estão alguns exemplos de funtores:

1. Ao tratar monoides como categorias por si só, é interessante entender o que seria equivalente a funtores nesse caso. Normalmente, as relações entre monoides são *homomorfismos de monoides*. Sejam dois monoides $(M, e, \cdot)$ e $(N, e', \star)$, um homomorfismo $\phi : M \to N$ é um mapeamento tal que
$$\phi(m \cdot m') = \phi(m) \star \phi(m')$$
e
$$\phi(e) = e'$$
É possível ver que ϕ possui a estrutura de funtor entre monoides. ϕ seria contravariante se $\phi(m \cdot m') = \phi(m') \star \phi(m)$

2. Existe um funtor $\mathtt{Core} : \mathbf{Mon} \to \mathbf{Grp}$ que pega um monoide e retorna um subconjunto desse monoide que possui elementos inversos, o que faz com que o monoide se torne um grupo, chamado de *cerne* (*Core*) do monoide.

3. Existe um funtor U : **Grp** $\to$ **Mon** chamado de *Forgetful functor* (Funtor esquecido) que pega um grupo e retorna o monoide correspondente, "esquecendo" a estrutura a mais que caracteriza os grupos.

4. Outro funtor esquecido é U : **Cat** $\to$ **Grph** que pega cada categoria e retorna o grafo correspondente a ela. Um Grafo $G = (V, A, s, t)$ é composto de um conjunto de vertices V, um conjunto de arestas A que são direcionadas, e um par de funções $s, t : A \to B$ que codifica a direção das arestas ao assinalar a cada aresta $a \in A$ um inicio $s(a) \in V$ e um fim $t(a) \in V$.

A coleção de objetos de uma categoria $\mathcal{C}$ será denotada por $\mathcal{C}_0$ e a coleção de morfismos será denotada por $\mathcal{C}_1$ na próxima definição:

Definição 8.16 ((AWODEY, 2010))**.** Um funtor $F : \mathcal{C} \to \mathcal{D}$ é dito:

- *injetivo em objetos* se a parte de objetos $F_0 : \mathcal{C}_0 \to \mathcal{D}_0$ é injetiva, é *sobrejetora em objetos* se F_0 é sobrejetora
- De forma similar, F é *injetiva* (resp. *sobrejetora*) em *setas* se a parte de setas $F_1 : \mathcal{C}_1 \to \mathcal{D}_1$ é injetiva (resp. sobrejetora)
- F é dito *fiel* (Faithful) se, para todo $A, B \in \mathcal{C}_0$, o mapa $F_{A,B} : \mathrm{Hom}_{\mathcal{C}}(A, B) \to \mathrm{Hom}_{\mathcal{D}}(FA, FB)$ definido por $f : F(f)$ é injetivo
- Similarmente F é dito *cheio* (full) se $F_{A,B}$ é sempre sobrejetor

Exemplo: Seja o funtor esquecido U : **Grp** $\to$ **Set**. U_0 é basicamente o mapeamento dos conjuntos que forma o grupo para os próprios conjuntos, logo U_0 é injetivo e sobrejetor em objetos. U_1 mapeia os homomorfismos de grupo para as funções correspondentes. Mas dois homomorfismos de grupo com o mesmo domínio e codomínio são iguais se são dados pelas mesmas funções nos conjuntos internos. Porém, nem todas as funções em **Set** são mapeadas por homomorfismos, logo U_1 é injetivo em setas mas não sobrejetor em setas. Os motivos para dizer que U_1 é injetor em setas podem ser usados para mostrar que U é fiel.

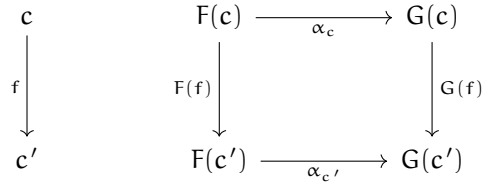
8.4 Transformações Naturais

Uma vez tendo definido morfismos entre categorias, se torna possível pensar morfismos entre esses morfismos. No caso, morfismos entre funtores:

Definição 8.17 (Transformação Natural, (ROSIAK, 2022))**.** Sejam duas categorias $\mathcal{C}$ e $\mathcal{D}$ e funtores $F, G : \mathcal{C} \to \mathcal{D}$. Uma *Transformação Natural* $\alpha : F \Rightarrow G$ representado em relação a seus dados como: Consiste no seguinte:

- Para cada objeto $c \in \mathcal{C}$, um morfismo $\alpha_c : F(c) \to G(c)$ em $\mathcal{D}$ chamado de c-componente de α, a coleção do qual (para todo objeto em $\mathcal{C}$) define os *componentes* da transformação natural

- Para cada morfismo $f : c \to c'$ em $\mathcal{C}$ o seguinte quadrado de morfismos, chamado de *quadrado de naturalidade* de f, que deve comutar em $\mathcal{D}$:



A coleção de transformações naturais entre F e G é por vezes denotada por $\mathbb{N}at(F, G)$

É dito que morfismos em uma categoria possuem *naturalidade* quando possuem um comportamento parecido com o do *quadrado de naturalidade*, ou seja se $G(f) \circ \alpha_c = \alpha_{c'} \circ F(f)$.

Uma vez que as transformações naturais ajudam a comparar dois funtores entre si, é interessante saber quando os dois funtores são praticamente iguais. Para isso, vamos usar a seguinte definição:

Definição 8.18 (Isomorfismo natural, (ROSIAK, 2022))**.** Um *isomorfismo natural* é uma transformação natural $\alpha : F \Rightarrow G$ para qual todo componente $\alpha_c : F(c) \to G(c)$ em $\mathcal{D}$ é um isomorfismo (na categoria alvo). Ou seja, cada α_c possui um inverso $\alpha_c^{-1} : G(c) \to F(c)$ onde os inversos formam componentes de uma transformação natural α^{-1} de G para F.

Se α for um isomorfismo, usa-se a notação $\alpha : F \cong G$

Uma vez definida a equivalência entre funtores, é interessante definir equivalência entre categorias:

Definição 8.19 (equivalência de categorias, (ROSIAK, 2022))**.** Uma *equivalência de categorias* consiste de um par de funtores $F : \mathcal{C} \to \mathcal{D}$ e $G : \mathcal{D} \to \mathcal{C}$ junto com os isomorfismos naturais $\eta : id_{\mathcal{C}} \cong G \circ F$ e $\epsilon : F \circ G \cong id_{\mathcal{D}}$. Outro jeito de dizer isso é que os funtores são inversos entre si "até o isomorfismo natural de funtores". As categorias $\mathcal{C}$ e $\mathcal{D}$ são ditas *equivalentes* se existe uma equivalência de categorias entre elas, isso é denotado por $\mathcal{C} \simeq \mathcal{D}$

Uma construção interessante é a categoria de setas que possui funtores como objetos e transformações naturais como morfismos, definida como:

Definição 8.20 ((ROSIAK, 2022))**.** Para qualquer par fixo de categorias $\mathcal{C}$ e $\mathcal{D}$, pode-se formar uma *categoria de funtores* denotada por $\mathcal{D}^{\mathcal{C}}$ (Ou também $\mathbb{F}un(\mathcal{C}, \mathcal{D})$) que possui:

- objetos: todos os funtores de $\mathcal{C}$ para $\mathcal{D}$
- morfismos: todas as transformações naturais entre tais funtores

Para demonstrar o aspecto de morfismo das transformações naturais, é necessário definir a transformação natural identidade, dada simplismente por $id_F : F \Rightarrow F$, e a composição entre transformações naturais, dada pela seguinte definição:

Definição 8.21 ((ROSIAK, 2022))**.** Sejam $\alpha : F \Rightarrow G$ e $\beta : G \Rightarrow H$ transformações naturais entre os funtores paralelos F, G, H entre $\mathcal{C}$ e $\mathcal{D}$ como no seguinte diagrama: Existe uma transformação natural $\beta \circ \alpha : F \Rightarrow H$, definida em cada componente como: $(\beta \circ \alpha)_c := \beta_c \circ \alpha_c$ dada pela composição de β e α.

Esse estilo de composição é denominado de *compisição vertical*

Já a composição horizontal denotada pelo simbolo $\diamond$ dado por $\beta \diamond \alpha : F_2 \circ F_1 \Rightarrow G_2 \circ G_1$, os quais cada componente em c de $\mathcal{C}$ é definido como o composto do seguinte diagrama comutativo:

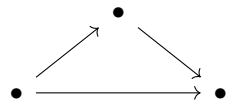
8.5 Limites

Antes de definir limites, é necessário definir o que são diagramas em uma categoria:

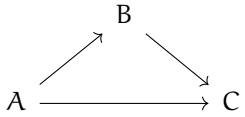
Definição 8.22 (Diagrama, (RIEHL, 2017))**.** Um *Diagrama* em uma categoria $\mathcal{C}$ é um funtor $F : \mathcal{J} \to \mathcal{C}$ que possui como domínio, chamado de *Categoria Índice* ou *template*, uma categoria pequena

Um diagrama é normalmente pensado como um grafo orientado. Um diagrama pode ser pensado como a instanciação ou realização em $\mathcal{C}$ de um template específico $\mathcal{J}$.

Seja



O template do seguinte diagrama:



na categoria $\mathcal{C}$. Seja qualquer objeto X em $\mathcal{C}$, existe um *Funtor constante* denotado também por X que leva todo objeto para X e todo morfismo para a seta identidade em X. Então X é, em si, um diagrama $X : \mathcal{J} \to \mathcal{C}$.

Como X e F são dois funtores, então vai pode ser construida uma transformação natural entre eles. Essa tranformação natural vai consistir na coleção de morfismos em X e objetos encontrados no diagrama F tal que esses morfismos comutam com os morfismos encontrados no diagrama. Quando esses morfismos vão de X para F, a construção é chamada de *Cone* e, no exemplo é da seguinte forma:

Já indo os morfismos de F para X, essa construção é chamada de *Cocone* e, seguindo o exemplo anterior, é da seguinte forma:

Um limite em um diagrama F é um caso especial de cone sobre F, onde o cone se aproxima o máximo possível do diagrama F.

Para definir um limite, é necessário fazer algumas definições primeiro:

Definição 8.23 (Objetos terminais e iniciais, (AWODEY, 2010))**.** Em uma categoria $\mathcal{C}$, um objeto

- 0 é dito *inicial* se para qualquer objeto C de $\mathcal{C}$ existe um morfismo único

$$0 \to C$$

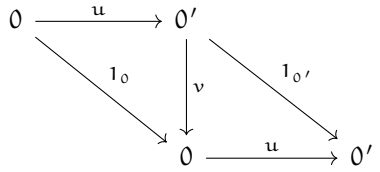
- 1 é dito *terminal* se para qualquer objeto C existe um morfismo único

$$C \to 1$$

Proposição 8.1 ((AWODEY, 2010))**.** Objetos iniciais (terminais) são únicos a menos que isomorfismo

Prova: Se C e C' são ambos iniciais (finais), então existe um isomorfismo *único* $C \to C'$.

Sejam 0 e $0'$ ambos objetos iniciais e seja o seguinte diagrama:



é possível ver que $u \circ v = 1_0$ e $v \; circ u = 1_{0'}$, logo eles são unicamente isomórficos. □

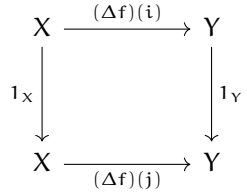
Exemplos:

- Em **Sets**, o conjunto vazio é o conjunto inicial e o conjunto unitário é o conjunto terminal. Ou seja, **Sets** possui um único conjunto inicial mas vários conjuntos terminais.
- Em **Cat**, a categoria **0** (com nenhum objeto nem nenhum morfismo) é inicial, enquanto a categoria **1** é terminal.
- Em **Grp** o grupo com único elemento é tanto inicial quanto terminal, assim como em **Mon**

Seja $t : \mathcal{J} \to \mathbf{1}$ o funtor único para a categoria terminal. Seja uma categoria $\mathcal{C}$ com um objeto $X \in Ob(\mathcal{C})$. Esse objeto pode ser representado pelo funtor $X : \mathbf{1} \to \mathcal{C}$. Então fazendo a composição de X em t, tem-se $X \circ t : \mathcal{J} \to \mathcal{C}$ que da o *funtor constante* em X, que envia cada objeto em $\mathcal{J}$ para o mesmo $\mathcal{C}$-objeto X e cada morfismo em $\mathcal{J}$ para a identidade 1_X naquele objeto. Ou seja, essa composição induz um funtor $[C \cong Fun(\mathbf{1}, \mathcal{C})] \to Fun(\mathcal{J}, \mathcal{C})$ denotado por $\Delta_t : \mathcal{C} \to Fun(\mathcal{J}, \mathcal{C}) = \mathcal{C}^{\mathcal{J}}$. No total, isso dá $\Delta : C \to \mathcal{C}^{\mathcal{J}}$ que leva cada objeto X para o funtor constante em X e cada morfismo $f : X \to Y$ à *transformação natural constante*. Seja $f : X \to Y$ em $\mathcal{C}$, então existe uma transformação natural $\Delta(X) \xrightarrow{\Delta(f)} \Delta(Y)$ tal que:

$$\begin{array}{ccc} (\Delta X)(i) & \xrightarrow{(\Delta f)(i)} & (\Delta Y)(i) \\ {\scriptstyle (\Delta X)(e)}\downarrow & & \downarrow{\scriptstyle (\Delta Y)(e)} \\ (\Delta X)(j) & \xrightarrow[(\Delta f)(j)]{} & (\Delta Y)(j) \end{array} \qquad \begin{array}{c} i \\ {\scriptstyle e}\downarrow \\ j \end{array}$$

comuta para cada aresta e da categoria índice $\mathcal{J}$. Mas como Δ leva para o funtor constante, que leva os objetos neles prórios, então os objetos nesse diagrama se reduzem a:



que obviamente comuta.

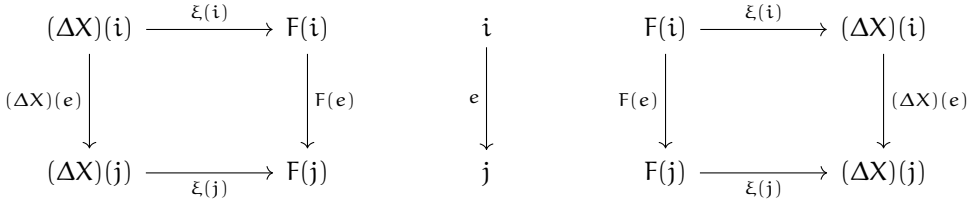
Considerando um $\mathcal{J}$-diagrama qualquer $F : \mathcal{J} \to \mathcal{C}$ e para cada $X \in \mathcal{C}$, as setas (que são transformações naturais):

$$\Delta X \to F \qquad F \to \Delta X$$

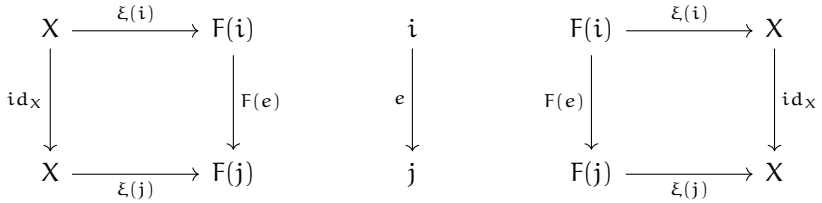
Então uma seta em $\mathcal{C}^{\mathcal{J}}$ que corresponde a essas setas é somente uma transformação natural, ou seja uma familia de setas em $\mathcal{C}$,

$$(\Delta X)(i) \xrightarrow{\xi(i)} F(i) \qquad F(i) \xrightarrow{\xi(i)} (\Delta X)(i)$$

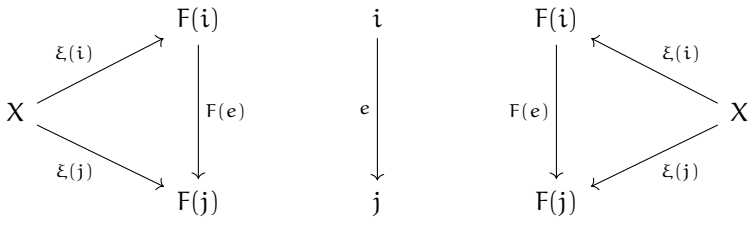
indexada pelos vários objetos ou nós em $\mathcal{J}$ e tais que:



Ou, fazendo a mesma redução anterior:



Mas isso se reduz aos seguintes triangulos comutativos:



As transformações naturais representadas pelos triangulos na esquerda dão *soluções esquerdas* ao diagrama em $\mathcal{C}$, as vezes chamadas de *Cones acima* do diagrama F com vertice *cume* em X. As transformações naturais representada pelos triangulos a direita dão as *soluções direitas* para o diagrama, também chamadas de *Cocones abaixo* do diagrama F com *nadir* (ponto mais baixo) em X.

Com isso, é possível definir a categoria de cones, onde um objeto na categoria de cones sobre F será um cone acima de F, com algum cume, enquanto o morfismo de um cone $\xi : X \Rightarrow F$ para um cone $\mu : Z \Rightarrow F$ é um morfismo $f : X \to Z$ em $\mathcal{C}$ tal que para cada indice $j \in \mathcal{J}$, $\mu_j \circ f = \xi_j$, ou seja, um mapa entre cumes tal que cada perna do cone domínio é fatorado através da perna correspondente do cone codominio.

A categoria de cones pode ser vista como a categoria slice Δ/F.

Usando essas noções, é possível definir o *limite* de F em termos do cone universal, onde um cone $\alpha : L \to F$ com vertice L é universal em respeito a F caso para cada cone $\Delta X \to F$ existe um mapa único $g : \Delta X \to L$ tal que o diagrama:

Comuta. Em uma definição formal:

Definição 8.24 ((AWODEY, 2010))**.** Um *limite* para o diagrama $F : \mathcal{J} \to \mathcal{C}$ é um objeto terminal $\lim F$ na categoria de cones em F, denotada por $\mathbf{Cone}(F)$. Se $\mathcal{J}$ for finito, o limite é chamado de *limite finito*

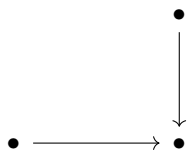
De forma dual, na categoria dos cocones $\mathbf{CoCones}(F)$ o cocone universal surge como *colimite* do diagrama F, denotado por $\operatorname{colim} F$:

Definição 8.25 ((ROSIAK, 2022))**.** O *colimite* do diagrama $F : \mathcal{J} \to \mathcal{C}$ é um objeto $\operatorname{colim} F$ em $\mathcal{C}$ junto com uma transformação natural $\epsilon : F \Rightarrow \operatorname{colim} F$ que satisfaz que para qualquer objeto X e qualquer transformação natural $\beta : F \Rightarrow X$ existe um morfismo único $h : \operatorname{colim} F \to X$ tal que $\beta = h \circ \epsilon$

Exemplos de limites:

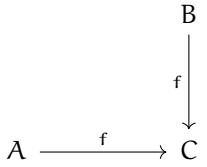
Em **Set**, o limite do diagrama discreto consistindo de conjuntos $X_1, X_2, \ldots,$ é o *Produto cartesiano* $\Pi_{i \in I} x_i$, onde essa construção vem com mapas projetivos $\pi_i : \Pi_{i \in I} X_i \to X_i$ para cada fator.

Seja o diagrama na forma:

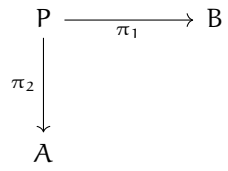


em uma categoria $\mathcal{C}$, o limite de um diagrama de tal formato é chamado de *pullback* (ou *produto fibrado*), consistindo de um objeto junto com morfismos que satisfazem essa propriedade universal. De forma formal:

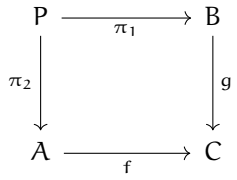
Definição 8.26. (AWODEY, 2010) Em uma categoria $\mathcal{C}$, um *pullback* de setas f, g com $\operatorname{cod}(f) = \operatorname{cod}(g)$



consiste de setas



tais que $f\pi_1 = g\pi_2$, ou seja o diagrama

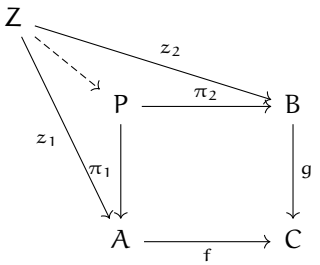


comuta.

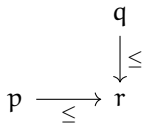
E também π_1 e π_2 são universais com essa propriedade. Ou seja sendo quaisquer $z_1 : Z \to A$ e $z_2 : Z \to B$, com $fz_1 = gz_2$, então existe um morfismo único

$$u : Z \to P$$

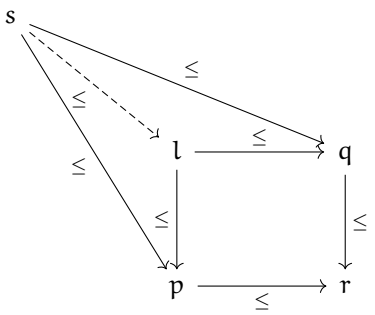
com $z_1 = \pi_1 u$ e $z_2 = \pi_2 u$. O diagrama equivalente é:



Considerando um poset como uma categoria, um diagrama pullback:



será dado por um elemento l, com $l \leq p$ e $l \leq q$, tais que para qualquer elemento s que também $s \leq p$ e $s \leq q$, tem-se que $s \leq l$:

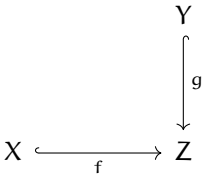


Ou seja, l é a *maior cota inferior* de p e q
Para **Sets** existem várias formas de construir pullbacks ((ROSIAK, 2022)):

- Seja $Z = \{*\}$ um conjunto unitário. ENtão como $\{*\}$ é o objeto terminal em **Sets**, tanto $X \xrightarrow{f} \{*\}$ e $Y \xrightarrow{g} \{*\}$ são funções únicas que levam tudo para $\{*\}$. O pullback de tal diagrama seria todos os pares (x, y) tais que tanto x e y são enviados para $\{*\}$ pelas funções únicas f e g, mas como não existem pares que satisfaçam esse requisito, é possível recuperar *todos* os pares (x, y) fazendo o pullback um diagrama que possui o conjunto inteiro $X \times Y$, chamado de produto cartesiano binário.

- Agora seja $Y = \{*\}$, enquanto Z é qualquer conjunto. Uma função $\{*\} \xrightarrow{g} Z$ só pega um elemento $z \in Z$. Então, para qualquer função $X \xrightarrow{f} Z$, o pullback será o subconjunto de elementos em X que são enviados para z através de f, recuperando a *pre-imagem* (ou *fibra*) de f em g.

- Agora sejam X e Y subconjuntos de Z, fazendo que f e g sejam inclusões, da forma:

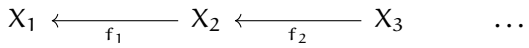


  Nesse caso, o pullback vai consistir em pares (x, y) tais que x e y são *iguais* na inclusão em Z. Ou seja, o pullback consiste em elementos $x = y$ de X que também estão em Y. Isso é a construção da *interseção* $X \cap Y$

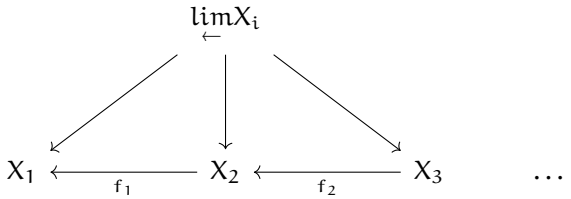
Seja um diagrama da forma

Em uma categoria $\mathcal{C}$, o limite de tal diagrama é chamado de *limite inverso*. Esse limite possui um objeto junto com os morfismos daquele objeto para cada $\bullet$ tal que todos os triangulos resultantes comutem e a propriedade universal do limite seja satisfeita.

Sejam $X_1, X_2, \ldots$ objetos em $\mathcal{C}$, então o limite inverso, denotado $\varprojlim X_i$ do diagrama



seria um objeto que mapeia em cada X_i na forma:



Em **Set**, esse seria um subconjunto $\varprojlim X_i$ do produto $\prod_{i\in I} X_i$ contendo todas as sequências $(x_1, x_2, x_3, \dots)$ onde o i-ésimo fator é tal que $f_i(x_{i+1}) = x_i$

Exemplo ((ROSIAK, 2022)): Seja um diagrama indexado por uma categoria consistindo de dois objetos e dois morfismos paralelos não identidade,

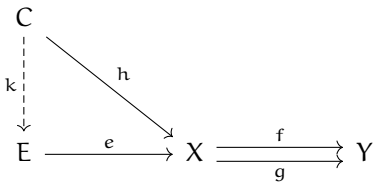
$$\bullet \rightrightarrows \bullet$$

Um diagrama em $\mathcal{C}$ de tal forma é um par de morfismos

$$X \underset{g}{\overset{f}{\rightrightarrows}} Y$$

em $\mathcal{C}$. Um cone com cume C consiste em de pares de morfismos $h : C \to X$ e $i : C \to Y$ tais que $f \circ h = i$ e $g \circ h = i$ junto com $f \circ h = g \circ h$. Ou seja, um cone acima desse par é um morfismo $h : C \to X$ tal que $f \circ h = g \circ h$.

A partir disso, é possível definir um objeto E junto com $e : E \to X$, chamado de *equalizador* de f e g, como a seta universal com a mesma propriedade, ou seja, $f \circ e = g \circ e$. Essa propriedade universal aponta que dado qualquer $h : C \to X$ tal que $f \circ h = g \circ h$ existe uma seta única $k : C \to E$ que fatora o morfismo h através de e tal que $e \circ k = h$ como no diagrama:



Em **Set**, o equalizador de f e g é o subconjunto de elementos de X para os quais as duas funções coincidem:

$$E = Eq(f, g) := \{x \in X | f(x) = g(x)\}$$

A seguinte definição mostra quando é possível "cancelar" setas em um lado:

Definição 8.27 (monomorfismo, (ROSIAK, 2022))**.** Um morfismo $i : B \to C$ em uma categoria é chamado de *monomorfismo* (ou morfismo *mônico*) se para qualquer A com morfismos paralelos f e g tais que:

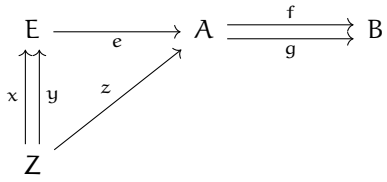
$$A \underset{g}{\overset{f}{\rightrightarrows}} B \xrightarrow[i]{} C$$

$i \circ f = i \circ g$ implica que $f = g$

monomorfismos generalizam a noção de funções injetoras em **Set**.

Proposição 8.2 ((AWODEY, 2010))**.** Em qualquer categoria, se $e : E \to A$ é um equalizador de um par de setas, então e é mônico

Prova: Considere o diagrama:



onde e é equalizador de f e g. Suponha que $ex = ey$, o que quer se mostrar é que $x = y$. Para isso, vamos usar a comutatividade do diagrama triangular. Seja $z = ex = ey$, então $fz = fex = gex = gz$, então existe um morfismo único $u : Z \to E$ tal que $eu = z$. Mas $ex = z$ e $ey = z$, então se segue que $x = u = y$. □

Definição 8.28 ((AWODEY, 2010))**.** Para mapas r e s, em qualquer categoria, r é chamado de *retração* de s caso $r \circ s$ for um mapeamento identidade. Nessa situação, s é chamado de *seção* de r

Exemplos de colimites

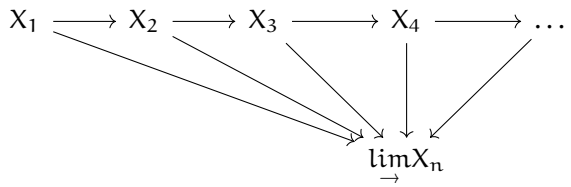
Em **Set**, o colimite de um diagrama discreto consistindo de conjuntos $X_1, X_2, \ldots$ é a *união disjunta* $\sqcup_{i\in I} X_i$, construida a partir de funções injetivas de cada X_i no conjunto coproduto $X = \sqcup_{i\in I} X_i$. Se os conjuntos são disjuntos par a par, a união disjunta se torna a união padrão $\cup$.

Em um poset, o colimite de um diagrama discreto é o seu *supremo* (ou *menor cota superior*) $\bigvee_{i\in I} p_i$.

O dual de limites inversos são *limites diretos*, que são o colimite do diagrama indexado pela categorial ordinal ω. Ou seja, para o diagrama

$$X_1 \longrightarrow X_2 \longrightarrow X_3 \longrightarrow X_4 \longrightarrow \ldots$$

seu colimite é o limite direto $\varinjlim X_n$, que define o diagrama de forma $\omega + 1$:

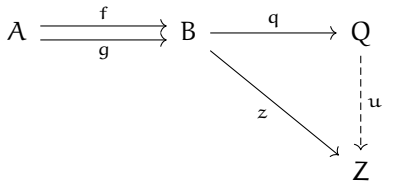


O colimite de uma sequência de conjuntos formada por inclusões:

$$X_1 \hookrightarrow X_2 \hookrightarrow X_3 \hookrightarrow X_4 \hookrightarrow \ldots$$

é a sua união $\bigcup_{n\geq 0} X_n$

Definição 8.29 (Coequalizador, (AWODEY, 2010))**.** Sejam duas setas paralelas $f, g : A \to B$ em uma categoria $\mathcal{C}$, um *coequalizador* consiste de Q e $q : B \to Q$ universais com a propriedade $qf = qg$ como no diagrama:



Ou seja, dado qualquer objeto Z e $z : B \to Z$, se $zf = zg$, então existe um morfismo único $u : Q \to Z$ tal que $uq = z$

Em **Set**, o coequalizador de duas funções $f, g : X \to Y$ é o *quociente* de Y pela menor relação de equivalência $\sim$ tal que para todo $x \in X$, $f(x) \sim g(x)$

Definição 8.30 ((ROSIAK, 2022)). $f : X \to Y$ é dito *epimorfismo* (ou somente *epi*) se para todo B e morfismos $h, h' : Y \to B$,

$$X \xrightarrow{f} Y \underset{h'}{\overset{h}{\rightrightarrows}} B$$

$h \circ f = h' \circ f$ implica que $h = h'$

Epimorfismos podem ser vistos como generalizações das funções sobrejetoras em **Set**.

Proposição 8.3 ((AWODEY, 2010)). se $q : B \to Q$ é um coequalizador de um par de setas, então q é epi.

Definição 8.31 ((ROSIAK, 2022)). Seja $\mathcal{C}$ uma categoria e seja $F : \mathcal{C} \to$ **Set** um funtor covariante. Então a *categoria de elementos* de F denotada $\int_{\mathcal{C}} F$ (ou somente $\int F$ se o contexto for claro) é definida:

- $Ob(\int F) = \{(c, x) | c \in \mathcal{C}, x \in F(c)\}$
- $Hom_{\int F}((c, x), (c', x')) = \{f : c \to c' | F(f)(x) = x'\}$

De forma dual, para o caso contravariante: para $F : \mathcal{C}^{op} \to$ **Set**, a *categoria de elementos* de F denotada $\int_{\mathcal{C}^{op}} F$ (ou somente $\int F$) é definida por:

- $Ob(\int F) = \{(c, x) | c \in \mathcal{C}, x \in F(c)\}$
- $Hom_{\int F}((c, x), (c', x')) = \{f : c \to c' | F(f)(x') = x\}$

Associado com essa construção estão os funtores $\pi_F : \int F \to \mathcal{C}$, chamados de *funtores de projeção*, que mandam cada objeto $(c, x) \in Ob(\int F)$ para o objeto $c \in Ob(\mathcal{C})$ ou $Ob(\mathcal{C}^{op})$ e cada morfismo $f : (c, x) \to (c', x')$ para o morfismo $f : c \to c'$.

Definição 8.32. Para qualquer classe de diagramas $K : \mathcal{J} \to \mathcal{C}$ em $\mathcal{C}$, um funtor $F : \mathcal{D} \to \mathcal{C}$ *preserva limites* se para qualquer diagrama K e cone limite sobre K, a imagem desse cone sob a ação do funtor definite um cone limite sobre o diagrama composto $F \circ K : \mathcal{J} \to \mathcal{D}$.

Definição 8.33 ((ROSIAK, 2022)). Um funtor é dito (co)*continuo* se ele preserva dos os (co)*limites* pequenos

8.6 Categorias Cartesianas Fechadas

8.6.1 Exponenciais

Primeiro, é necessário uma digreção dentro da categoria dos conjuntos.

Seja a função entre conjuntos

$$f(x, y) : A \times B \to C$$

escrita usando variáveis $x \in A$ e $y \in B$.

Se $a \in A$ for mantido constante, então tem-se a função

$$f(a, y) : B \to C$$

e então o elemento

$$f(a, y) \in C^B$$

do conjunto C^B de todas as funções de B para C.

Se a for variado, então pode-se criar outra função

$$\bar{f} : A \to C^B$$

que leva como parâmetro a para a função $f_a(y) : B \to C$.

A relação entre essas funções pode ser determinada pela seguinte equação:

$$\bar{f}(a)(b) = f(a, b)$$

Ou seja, existe um isomorfismo entre conjuntos:

$$Sets(A \times B, C) \cong Sets(A, C^B)$$

Ou seja, existe uma correspondência bijetiva entre essas funções mediada por uma operação de *avaliação* (*evaluation*):

$$eval : C^B \times B \to C$$

dada por:

$$eval(g, b) = g(b)$$

É possível definir exponênciais em qualquer categoria na seguinte forma:

Definição 8.34 ((AWODEY, 2010))**.** Seja a categoria $\mathcal{C}$ que possui produtos binários. Um *exponencial* de objetos B e C de $\mathcal{C}$ consiste em um objeto

$$C^B$$

e uma seta

$$\epsilon : C^B \times B \to C$$

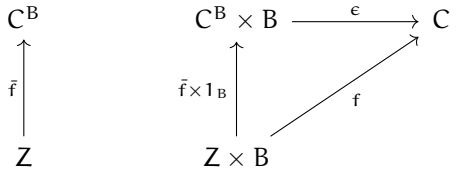
tal que, para todo objeto Z e seta $f : Z \times B \to C$ existe uma seta única

$$\bar{f} : Z \times C^B$$

tal que

$$\epsilon \circ (\bar{f} \times 1_B) = f$$

como no diagrama:



8.6.2 Categorias Cartesianas Fechadas

Definição 8.35 ((AWODEY, 2010))**.** Uma categoria $\mathcal{C}$ é dita ter *todos os produtos finitos* se ele possui um objeto terminal e todos os produtos binários (e também produtos de qualquer cardinalidade finita). A categoria $\mathcal{C}$ possui *todos os produtos pequenos* se todo conjunto de objetos em $\mathcal{C}$ possui produtos

Com isso, é possível definir categorias cartesianas fechadas da seguinte forma:

Definição 8.36 ((AWODEY, 2010))**.** Uma categoria é chamada *cartesiana fechada* se possui todos os produtos finitos e exponenciais

Part IV
Teorias Homotópicas de Tipos

9 Teoria dos Tipos de Martin-Löf

Nesse capítulo, o interesse é dar uma revisão geral da teoria dos tipos dependentes e apresentar a Teoria dos tipos de Martin-Löf, uma teoria dos tipos dependentes com tipos adicionais.

9.1 Apontamentos iniciais

Como visto no capítulo 5, a teoria dos tipos dependentes é uma extensão da teoria dos tipos simples que permite a construção de tipos que dependam de termos. Por exemplo, seja $n : \mathbf{N}$ um termo do tipo dos naturais, é possível construir o tipo Vect n dos vetores de tamanho n.

A teoria dos tipos dependentes de Martin-Löf foi chamada por ele também de Teoria dos Tipos Intuicionista, pois é um tipo de teoria dos tipos que se preocupa com a formalização da matemática intuicionista presente em autores como Bishop. Na apresentação da Teoria dos tipos de Martin-Lof (MLTT), serão feitas algumas mudanças de sintaxe presentes em textos mais recentes como (RIJKE, 2022), por exemplo a apresentação de termos como $a : A$ ao invés de $a \in A$

9.2 Os tipos da MLTT

A exposição dos tipos presentes na teoria dos tipos de Martin-Löf é bastante correlata à exposição dos tipos presentes na teoria dos tipos simples feita na segunda parte do capítulo 3. Como na teoria dos tipos simples, as regras de cada tipo da Teoria dos Tipos de Martin-Lof são divididas em quatro partes:

- Uma *regra de formação* que diz como formar o tipo
- Uma *regra de introdução* que diz como introduzir novos termos do tipo
- Uma *regra de eliminação* que diz como usar e remover termos do tipo
- Uma *regra de computação* que diz como as regras de introdução e de eliminação se relacionam

9.2.1 O tipo de função dependente

Sejam A um tipo e x um elemento desse tipo, $x : A$, um tipo B que depende de x é denotado por $B(x)$, esse tipo será habitado por um termo $b(x)$. Nesse caso, b pode ser visto como uma função que leva um termo $x : A$ a um termo $b(x) : B(x)$, chamada de função dependente. O tipo de todas as funções dependentes que levam um termo $x : A$ qualquer a um termo do tipo $B(x)$ é denotado por $\Pi_{x:A} B(x)$.

Essa apresentação pode ser formalizada pela seguinte regra de formação:

$$\frac{\Gamma, x : A \vdash B(x)\ \text{Type}}{\Gamma \vdash \Pi_{x:A} B(x)\ \text{Type}}\ \Pi\text{-form}$$

Um elemento $f : \Pi_{x:A} B(x)$ é uma função que leva termos $x : A$ a termos $b(x) : B(x)$, essa função pode ser representada por um λ-termo a partir da seguinte regra de introdução:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash \lambda x.b(x) : \Pi_{x:A} B(x)}\ \Pi\text{-intro}$$

Para eliminar um termo $f : \Pi_{x:A} B(x)$ é preciso somente computar o resultado da aplicação $f(x)$ para um $x : A$

$$\frac{\Gamma \vdash f : \Pi_{x:A} B(x)}{\Gamma, x : A \vdash f(x) : B(x)}\ \Pi\text{-intro}$$

As regras de computação para o tipo de função dependente são as regras de β e η-conversão:

$$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma \vdash (\lambda y.b(y))(x) = b(x) : B(x)}\ \beta$$

$$\frac{\Gamma \vdash f : \Pi_{x:A} B(x)}{\Gamma \vdash (\lambda x.f(x)) = f : \Pi_{x:A} B(x)}\ \eta$$

Uma vez definido o tipo de funções dependentes é possível definir o tipo de funções ordinárias como:

$$\frac{\Gamma \vdash A\ \text{Type} \qquad \Gamma \vdash B\ \text{Type}}{\Gamma \vdash A \to B = \Pi_{x:A} B(x)\ \text{Type}}$$

As regras para o tipo $A \to B$ são iguais as regras do tipo $\Pi_{x:A} B(x)$.

Uma função importante é a função identidade $\mathrm{id}_A : A \to A$, definida como $\lambda x.x : A \to A$.

Para funções dependentes que dependem de mais de um termo, é possível usar a notação

$$\Pi_{x:A} \Pi_{y:B} C(x, y)$$

mas também

$$\Pi_{x:A, y:B} C(x, y)$$

Definição 9.1 ((RIJKE, 2022))**.** Sejam A, B e C três tipos em um contexto Γ, existe uma operação de *composição*

$$\mathrm{comp} : (B \to C) \to ((A \to B) \to (A \to C))$$

escrita como $g \circ f$ para $\mathrm{comp}(g, f)$

O termo comp pode ser construido como:

$$\mathrm{comp} := \lambda g.\lambda f.\lambda x.g(f(x))$$

Lema 9.1 ((RIJKE, 2022))**.** A composição de funções é associativa, ou seja, é possível derivar

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash g : B \to C \qquad \Gamma \vdash h : C \to D}{\Gamma \vdash (h \circ g) \circ f = h \circ (g \circ f) : A \to D}$$

Prova: a ideia principal da prova é que tanto $(h \circ g) \circ f$ quanto $h \circ (g \circ f)$ são evaluados para $h(g(f(x)))$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B} \qquad \dfrac{\dfrac{\Gamma \vdash g : B \to C}{\Gamma, y : B \vdash g(y) : C}}{\Gamma, x : A, y : B \vdash g(y) : C}}{\Gamma, x : A \vdash g(f(x)) : C} \qquad \dfrac{\dfrac{\Gamma \vdash h : C \to D}{\Gamma, z : C \vdash h(z) : D}}{\Gamma, x : A, z : C \vdash h(z) : D}}{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash h(g(f(x))) : A \to D}{\Gamma, x : A \vdash h(g(f(x))) = h(g(f(x))) : A \to D}}{\Gamma, x : A \vdash ((h \circ g)) \circ f(x) = h \circ (g \circ f(x)) : A \to D}}{\Gamma, x : A \vdash ((h \circ g) \circ f)(x) = (h \circ (g \circ f))(x) : A \to D}}{\Gamma, x : A \vdash (h \circ g) \circ f = h \circ (g \circ f) : A \to D}}{\Gamma \vdash (h \circ g) \circ f = h \circ (g \circ f) : A \to D}$$

Lema 9.2 ((RIJKE, 2022))**.** A composição de funções satisfaz regras de unidade a esquerda e a direita, ou seja é possível derivar:

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \mathrm{id}_B \circ f = f : A \to B}$$

e

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \circ \mathrm{id}_A = f : A \to B}$$

Prova: Para a primeira regra:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A \text{ Type}}{\Gamma, x : A \vdash f(x) : B} \qquad \dfrac{\Gamma \vdash A \text{ Type} \qquad \dfrac{\Gamma \vdash B \text{ Type}}{\Gamma, y : B \vdash \mathrm{id}(y) = y : B}}{\Gamma, x : A, y : B \vdash \mathrm{id}(y) = y : B}}{\Gamma, x : A \vdash \mathrm{id}(f(x)) = (f(x)) : B}}{\Gamma \vdash \lambda x.\mathrm{id}(f(x)) = \lambda x.f(x) : A \to B} \qquad \dfrac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x; f(x) = f : A \to B}$$
$$\Gamma \vdash \mathrm{id} \circ f = f : A \to B$$

a segunda regra fica como exercício para o leitor

9.2.2 O tipo dos números naturais

O tipo dos números naturais $\mathbb{N}$ é o primeiro tipo indutivo que será apresentado aqui nessa seção. Ele é, como os outros tipos indutivos, um tipo **postulado**, ou seja, sua regra de formação é construida a partir do contexto vazio da seguinte forma:

$$\frac{}{\vdash \mathbb{N} \text{ Type}}\ \mathbb{N}\text{-form}$$

A construção dos elementos do tipo dos inteiros segue a construção dos axiomas de Peano: é postulado um elemento inicial que, na literatura da teoria dos tipos é sempre o 0, e uma função sucessor que permite construir novos elementos

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad \frac{}{\vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

A eliminação do tipo $\mathbb{N}$ representa o princípio da indução dos números naturais, onde o tipo resultante é o predicado $\forall_{n \in \mathbb{N}} P(n)$. Essa regra é:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \mathrm{type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S) : \Pi_{n:\mathbb{N}} P(n)}\ \mathbb{N}\text{-ind}$$

Da esquerda para a direita é possível ver que, sendo p_0 a prova que $P(0_{\mathbb{N}})$ é válido e p_S a prova que dado um termo de tipo $P(n)$ é sempre possível construir um termo $P(\mathrm{succ}_{\mathbb{N}}(n))$, as premissas seguem o axioma de peano para a indução nos números naturais. O termo $\mathrm{ind}_{\mathbb{N}}(p_0, p_S)$ pode ser visto também como uma função que recebe os termos p_0 e p_S como entrada, sendo representada de forma geral na seguinte regra equivalente:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \mathrm{type}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}} : P(0_{\mathbb{N}}) \to ((\Pi_{n:\mathbb{N}} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))) \to \Pi_{n:\mathbb{N}} P(n))}$$

Ou seja, para cada família de tipos P sobre $\mathbb{N}$ existe uma *função* $\mathrm{ind}_{\mathbb{N}}$ que pega dois argumentos, o primeiro sendo o caso base e o segundo o passo indutivo, e retorna uma seção de P

A regra de computação para $\mathbb{N}$ postula que a função dependente $\mathrm{nd}_{\mathbb{N}}(p_0, p_S) : \Pi_{n:\mathbb{N}} P(n)$ se comporta como experado quando aplicada a $0_{\mathbb{N}}$ ou ao sucessor.

Para a aplicação com o $0_{\mathbb{N}}$:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \mathrm{type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S, 0_{\mathbb{N}}) = p_0 : P(0_{\mathbb{N}})}$$

Para o sucessor:

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n)\ \mathrm{type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_S : \Pi_{n:\mathbb{N}} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_S, \mathrm{succ}_{\mathbb{N}}) = p_0 : P(\mathrm{succ}_{\mathbb{N}})}$$

Exemplo:

Definição 9.2 (Adição, (RIJKE, 2022))**.** Definindo uma função

$$\mathrm{add}_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

que satisfaça a especificação:

$$\mathrm{add}_{\mathbb{N}}(m, 0_{\mathbb{N}}) = m$$

$$\mathrm{add}_{\mathbb{N}}(m, \mathrm{succ}_{\mathbb{N}}(n)) = \mathrm{succ}_{\mathbb{N}}(\mathrm{add}_{\mathbb{N}}(m, n))$$

$\mathrm{add}_{\mathbb{N}}(m, n)$ será denotado por $m + n$

Para sua construção, é necessário usar a regra de eliminação para $P = \mathbb{N}$. Logo, as premissas se tornam:

$$m : \mathbb{N} \vdash \mathbb{N}$$
$$m : \mathbb{N} \vdash add - zero_{\mathbb{N}}(m) : \mathbb{N}$$
$$m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

e sua conclusão é:

$$m : \mathbb{N} \vdash add_{\mathbb{N}}(m) := ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}$$

O termo $add - zero_{\mathbb{N}}(m)$ é o próprio m, já para a adição do sucessor, é necessário saber qual o comportamento da soma para o sucesso, que é a seguinte:

$$add_{\mathbb{N}}(m, succ_{\mathbb{N}}(n)) = succ_{\mathbb{N}}(add_{\mathbb{N}}(m, n))$$

Logo, $add - succ_{\mathbb{N}}(m)$ é o mesmo que o sucessor da adição:

$$add - succ_{\mathbb{N}}(m) := \lambda n.succ_{\mathbb{N}}$$

A árvore de derivação desse termo é o seeguinte:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \mathbb{N} \to \mathbb{N} \qquad \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}{n : \mathbb{N} \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N}, n : \mathbb{N} \vdash succ_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N} \vdash \lambda n.succ_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}}{m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) := \lambda n.succ_{\mathbb{N}} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}$$

A regra de inferência final se torna:

$$\dfrac{\dfrac{\dfrac{\vdots}{m : \mathbb{N} \vdash \mathbb{N}\ \mathrm{Type}} \quad \dfrac{\vdots}{m : \mathbb{N} \vdash add - zero_{\mathbb{N}}(m) := m : \mathbb{N}} \quad \dfrac{\vdots}{m : \mathbb{N} \vdash add - succ_{\mathbb{N}}(m) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}}{m : \mathbb{N} \vdash ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}}{m : \mathbb{N} \vdash add_{\mathbb{N}}(m) := ind_{\mathbb{N}}(add - zero_{\mathbb{N}}(m), add - succ_{\mathbb{N}}(m)) : \mathbb{N} \to \mathbb{N}}$$

Outra forma de definir a adição é usando *casamento de padrões*, na seguinte forma:

$$add_{\mathbb{N}}(m, 0_{\mathbb{N}}) = m$$
$$add_{\mathbb{N}}(m, succ_{\mathbb{N}}(n)) = succ_{\mathbb{N}}(add_{\mathbb{N}}(m, n))$$

essa especificação é suficiente para descrever o comportamento de $add_{\mathbb{N}}(m)$ a partir dos construtores de $\mathbb{N}$.

Para uma função qualquer f, é possível definir seu comportamento em $\mathbb{N}$ através de

$$f(0_{\mathbb{N}}) = p_0$$
$$f(succ_{\mathbb{N}}(n)) = p_S(n, f(n))$$

Quando f é definida dessa forma, é dito que ela é definida por **casamento de padrões** sobre n

9.2.3 Outros tipos indutivos

O tipo dos números naturais é o caso mais emblemático dos tipos indutivos. Esses tipos seguem regras de introdução similares a $\mathbb{N}$. Sua estrutura é dividida em :

- Construtores que definem a estrutura do tipo através de seus elementos base
- Um princípio indutivo, através das regras de eliminação, que especifica o que é necessário para construir uma família de tipos arbitrária sobre o tipo.
- Regras de computação que definem como os termos gerados indutivamente se relacionam com os construtores.

O Tipo Unitário

Um exemplo básico de um tipo indutivo é o *tipo unitário*, que só possui um construtor.

Definição 9.3 ((RIJKE, 2022))**.** O **tipo unitário** é definido como o tipo $\mathbf{1}$ equipado com um termo

$$\star : \mathbf{1}$$

satisfazendo o princípio indutivo que para qualquer família de tipos $P(x)$ indexada por $x : \mathbf{1}$, existe uma função

$$ind_{\mathbf{1}} : P(\star) \to \Pi_{(x:\mathbf{1})} P(x)$$

para qual a regra de computação

$$ind_{\mathbf{1}}(p, \star) = p$$

é válida

Caso P não dependa de x o princípio da indução se torna

$$ind_{\mathbf{1}} : P \to (\mathbf{1} \to P)$$

Ou seja, nesse caso, para cada $x : P$ existe uma função $pt_x := ind_{\mathbf{1}}(x) : \mathbf{1} \to A$

O Tipo Vazio

O tipo vazio é um tipo indutivo com nenhum construtor e, por não ter nenhum construtor, não possui regras de computação.

Definição 9.4 ((RIJKE, 2022))**.** O **tipo vazio** é o tipo $\varnothing$ que satisfaz o principio de indução que para qualquer família de tipos $P(x)$ indexada por $x : \varnothing$ existe um termo

$$ind_{\varnothing} : \Pi_{x:\varnothing} P(x)$$

Quando P não depende de x, se segue o seguinte termo na indução:

$$ex - falso := ind_{\varnothing} : \varnothing \to P$$

Essa função pode ser usada para mostrar qualquer conclusão a partir de uma contradição. Usando a interpretação de proposições como tipos, é possível construir operadores lógicos a partir desse tipo vazio como a negação:

Definição 9.5 ((RIJKE, 2022))**.** Para qualquer tipo A, a **negação** de A é definida como:

$$\neg A := A \to \varnothing$$

Um tipo A é dito **vazio** se vem equipado com um elemento do tipo $\mathtt{neg}A$, ou seja:

$$\mathtt{is-empty}(A) := A \to \varnothing$$

Exemplo ((RIJKE, 2022)): para quais quer dois tipos P e Q existe uma função

$$(P \to Q) \to (\neg Q \to \neg P)$$

Prova: Pela regra da abstração, é possível assumir a função $f : P \to Q$. Com isso, é necessário definir uma função $(\neg Q \to \neg P)$. Mas novamente pela regra da abstração é possível assumir $\tilde{q} : Q \to \varnothing$. Com isso sobra para provar $\neg P$ que é o mesmo que $P \to \varnothing$, então é possível abstrair $p : P$ ficando como objetivo construir o termo $\varnothing$.

Dessa forma, usando a aplicação, $fp : Q$ e $\tilde{q}(f(p)) : \varnothing$. Logo, o termo final é:

$$\lambda f.\lambda \tilde{q}.\lambda p.\tilde{q}(f(p)) : (P \to Q) \to (\neg Q \to \neg P)$$

□

O tipo coproduto

Definição 9.6 ((RIJKE, 2022))**.** Sejam A e B tipos. O **tipo coproduto** $A + B$ é o tipo equipado com

$$\mathtt{inl} : A \to A + B$$

$$\mathtt{inr} : B \to A + B$$

satisfazendo o princípio de indução que para qualquer família de tipos $P(x)$ indexada por $x : A + B$ existe o termo

$$\mathtt{ind}_+ : (\Pi_{(x:A)} P(\mathtt{inl}(x))) \to ((\Pi_{(y:B)} P(\mathtt{inr}(y))) \to \Pi_{(z:A+B)} P(z))$$

Para o qual seguem as regras de computação

$$\mathtt{ind}_+(f, g, \mathtt{inl}(x)) = f(x)$$

$$\mathtt{ind}_+(f, g, \mathtt{inr}(y)) = g(y)$$

Notação: por vezes é escrito $[f, g]$ ao invés de $\mathtt{ind}_+(f, g)$
O coproduto de dois tipos pode ser chamado também de **soma disjunta**.
É possível derivar, sem as dependências, a função

$$\mathtt{ind}_+ : (A \to X) \to ((B \to X) \to (A + B \to X))$$

É interessante notar que, usando a interpretação de proposições como tipos, essa regra é similar à regra da eliminação do *ou* na dedução natural:

$$(P \to Q) \to ((R \to Q) \to (P \vee R \to Q))$$

Outra coisa a notar é que é possível construir a função

$$f + g : A + B \to A' + B'$$

para toda função $f : A \to A'$ e $g : B \to B'$ definida como:

$$(f + g)(inl(x)) := inl(f(x))$$
$$(f + g)(inr(x)) := inr(g(y))$$

Proposição 9.1 ((RIJKE, 2022))**.** Sejam dois tipos A e B e suponha que B é vazio, então existe uma função

$$(A + B) \to A$$

Essa proposição pode ser reescrita como: existe uma função

$$is - empty(B) \to ((A + B) \to A)$$

Prova: Pelo princípio indutivo do coproduto $A + B$ é possível ver que para provar $(A + B) \to A$ é necessário construir duas funções $f : A \to A$ e $g : B \to A$. Nesse caso, f é simplesmente a função identidade $id_A : A \to A$. Dizer que B é vazio é o mesmo que construir a função $\tilde{b} : B \to \varnothing$ e também é possível definir $ex - false :\to A$ e g se torna $g := ex - false \circ \tilde{b}$ □

O Tipo dos Inteiros

O conjunto dos inteiros pode ser visto, de forma inicial, como o conjunto dos números naturais unido com o conjunto dos números negativos. Essa visão pode ser codificada através da seguinte definição:

Definição 9.7 ((RIJKE, 2022))**.** O tipo dos **Inteiros** é definido como o tipo $\mathbb{Z} := \mathbb{N} + (\mathbf{1} + \mathbb{N})$. O tipo dos inteiros vem equipado com as funções de inclusão dos inteiros positivos e negativos:

$$in - pos := inr \circ inr : \mathbb{N} \to \mathbb{Z}$$
$$in - neg := inl : \mathbb{N} \to \mathbb{Z}$$

e com as constantes

$$-1_{\mathbb{Z}} := in - neg(0)$$
$$0_{\mathbb{Z}} := inr(inl(\star))$$
$$1_{\mathbb{Z}} := in - pos(0)$$

O princípio de indução para os inteiros afirma que, para qualquer família de tipos P sobre $\mathbb{Z}$, é possível definir uma função dependente $f : \Pi_{k:\mathbb{Z}} P(k)$ recursivamente como:

$$f(-1_{\mathbb{Z}}) := p_{-1}$$
$$f(in - neg(succ_{\mathbb{N}}(n))) := p_{-S}(n, f(in - neg(n)))$$
$$f(0_{\mathbb{Z}}) := p_0$$
$$f(1_{\mathbb{Z}}) := p_1$$
$$f(in - pos(succ_{\mathbb{N}}(n))) := p_S(n, f(in - pos(n)))$$

onde os tipos de p_{-1}, p_{-S}, p_0, p_1 e p_S são

$$p_{-1} : P(-1_{\mathbb{Z}})$$

$$p_{-S} : \Pi_{n:\mathbb{N}} P(in-neg(n)) \to P(in-neg(succ_{\mathbb{N}}(n)))$$
$$p_0 : P(0_{\mathbb{Z}})$$
$$p_1 : P(1_{\mathbb{Z}})$$
$$p_S : \Pi_{n:\mathbb{N}} P(in-pos(n)) \to P(in-pos(succ_{\mathbb{N}}(n)))$$

Definição 9.8 ((RIJKE, 2022))**.** A **função sucessora** dos inteiros $succ_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z}$ pode ser definida tomando:

$$succ_{\mathbb{Z}}(-1_{\mathbb{Z}}) := 0_{\mathbb{Z}}$$
$$succ_{\mathbb{Z}}(in-neg(succ_{\mathbb{N}}(n))) := in-neg(n)$$
$$succ_{\mathbb{Z}}(0_{\mathbb{Z}}) := 1_{\mathbb{Z}}$$
$$succ_{\mathbb{Z}}(1_{\mathbb{Z}}) := in-pos(1_{\mathbb{N}})$$
$$succ_{\mathbb{Z}}(in-pos(succ_{\mathbb{N}}(n))) := in-pos(succ_{\mathbb{N}}(succ_{\mathbb{N}}(n)))$$

O Tipo par dependente

Seja B uma família de tipos sobre A, é interessante considerar pares (a, b) de termos, onde $a : A$ e $b : B(a)$. Como nesse par b depende de a, ele é chamado de **par dependente**. O tipo dos pares dependentes é um tipo indutivo

Definição 9.9 ((RIJKE, 2022))**.** Seja B uma família de tipos sobre A. O **tipo par dependente**, também chamado de **Tipo** Σ, é definido como o tipo indutivo $\Sigma_{(x:A)}B(x)$ equipado com uma **função pareadora**

$$pair : \Pi_{x:A}(B(x) \to \Sigma_{(y:A)}B(y))$$

O princípio de indução para $\Sigma_{(x:A)}B(x)$ afirma que para qualquer família de tipos $P(p)$ indexada por $p : \Sigma_{(x:A)}B(x)$, existe uma função

$$ind_{\Sigma} : \Pi_{x:A}\Pi_{y:B(x)}P(pair(x, y)) \to (\Pi_{(z:\Sigma_{(x:A)}B(x))}P(z))$$

satisfazendo a regra de computação

$$ind_{\Sigma}(g, pair(x, y)) = g(x, y)$$

O princípio de indução de tipos Σ pode ser usado para definir as funções de projeção:

Definição 9.10 ((RIJKE, 2022))**.** Seja A um tipo e B uma família de tipos sobre A

1. O **mapa da primeira projeção**

$$pr_1(\Sigma_{(x:A)}B(x)) \to A$$

é definido por indução como

$$pr_1(x, y) := x$$

2. O **mapa da segunda projeção**

$$pr_2 : \Pi_{(p:Sigma_{(x:A)}B(x))}B(pr_1(p))$$

definido por indução como

$$pr_2(x, y) := y$$

O princípio da indução de Σ é uma forma de *Uncurrying*, pois ele pega uma função que aceita duas entradas

$$\lambda x.\lambda y.f(x, y) : \Pi_{x:A}\Pi_{y:B(x)}P(pair(x, y))$$

e retorna uma função que aceita um par de entradas

$$f : (\Pi_{(z:\Sigma_{(x:A)}B(x))}P(z))$$

A operação oposta também é possível de construir, chamada de *Currying*:

$$ev - pair : (\Pi_{(z:\Sigma_{(x:A)}B(x))}P(z)) \to (\Pi_{x:A}\Pi_{y:B(x)}P(pair(x, y)))$$

Um caso especial do tipo Σ ocorre quando B é uma família constante sobre A. Nesse caso, o tipo $\Sigma_{x:A}B$ é o tipo dos pares *ordinários* (x, y) onde $x : A$ e $y : B$, também chamado de **tipo produto**:

Definição 9.11 ((RIJKE, 2022))**.** Sejam dois tipos A e B. É possível definir o **tipo de produtos (cartesianos)** $A \times B$ de A e B como:

$$A \times B := \Sigma_{x:A}B$$

O tipo produto também satisfaz o principio de indução dos tipos Σ, mas sem a dependência em B, da seguinte forma:

$$ind_\times : \Pi_{x:A}\Pi_{y:B}P(x, y) \to (\Pi_{z:A\times B}P(z))$$

Os seus mapas projetivos também são definidos de forma similar aos mapas do tipo Σ. Usando a interpretação de proposições como tipos, $A \times B$ é a *conjunção*.

O Tipo dos booleanos

Outro tipo indutivo é o tipo dos booleanos:

Definição 9.12 ((RIJKE, 2022))**.** O tipo dos booleanos é o tipo indutivo $bool$ que vem equipado com:

$$false : bool \qquad\qquad true : bool \tag{1}$$

O princípio indutivo dos booleanos fala que para qualquer família de tipos $P(x)$ indexada por $x : bool$ existe um termo

$$ind - bool : P(false) \to (P(true) \to \Pi_{x:bool}P(x))$$

para o qual são válidas as seguintes regras de computação:

$$ind - bool(p_0, p_1, false) := p_0$$
$$ind - bool(p_0, p_1, true) := p_1$$

O Tipo das listas

Definição 9.13 ((RIJKE, 2022))**.** Para qualquer tipo A, é possível definir o tipo $\mathtt{list}(A)$ de listas de elementos de A como tipo indutivo com construtores

$$\mathtt{nil} : \mathtt{list}(A)$$

$$\mathtt{cons} : A \to (\mathtt{list}(A) \to \mathtt{list}(A))$$

O princípio indutivos das listas em A fala que para qualquer família de tipos $P(x)$ indexada por $x : \mathtt{list}(A)$ existe um termo:

$$\mathtt{ind} - \mathtt{list} - A : P(\mathtt{nil}) \to (\Pi_{l:\mathtt{list}A} P(\mathtt{cons}(A, l)) \to \Pi_{x:\mathtt{list}} P(x))$$

com as regras de computação:

$$\mathtt{ind} - \mathtt{list} - A(p_{\mathtt{nil}}, p_{\mathtt{cons}}, \mathtt{nil}) := p_{\mathtt{nil}}$$
$$\mathtt{ind} - \mathtt{list} - A(p_{\mathtt{nil}}, p_{\mathtt{cons}}, \mathtt{cons}(A, l)) := p_{\mathtt{cons}}$$

9.3 O Tipo identidade

Usando a interpretação de tipos como proposições, para provar uma proposição P é necessário formar um elemento p que seja do tipo P, na forma $p : P$. De forma similar, para provar a igualdade de dois termos $x, y : A$, é necessário demonstrar que existe um elemento p tal que $p : x = y$. Dessa forma, se torna necessário desenvolver um tipo que permita trabalhar com essas igualdades. Esse tipo será, como alguns outros vistos até então, um tipo indutivo.

Definição 9.14 ((RIJKE, 2022))**.** Considere um tipo A e $a : A$. Então é possível definir o **tipo identidade** de A em a como uma família indutiva de tipos $a =_A x$ indexada por $x : A$, com o construtor

$$\mathtt{refl}_a : a =_A a$$

O princípio indutivo do tipo identidade postula que para qualquer família de tipos $P(x, p)$ indexada por $x : A$ e $p : a =_A x$, existe uma função

$$\mathtt{ind} - \mathtt{eq}_a : P(a, \mathtt{refl}_a) \to \Pi_{x:A} \Pi_{p:a=_A x} P(x, p)$$

satisfazendo

$$\mathtt{ind} - \mathtt{eq}(u, a, \mathtt{refl}_a) = u$$

dado $u : P(a, \mathtt{refl}_a)$

Um elemento do tipo $a =_A x$ é chamado de **identificação** de a com x, por vezes também chamado de **caminho** de a para x. O princípio indutivo dos tipos identidade é por vezes chamado de **eliminação da identificação** ou **indução de caminho**.

9.3.1 Operações básicas

Uma vez definido o tipo identidade, é interessante mostrar que as identidades podem ser *concatenadas* e *invertidas*, que correspondem à transitividade e a simetria do tipo identidade.

Definição 9.15 ((RIJKE, 2022))**.** Seja A um tipo. A operação de **concateção** é definida como:

$$concat : \Pi_{x,y,z:A}(x = y) \to ((y = z) \to (x = z))$$

também é possível escrever $p \centerdot q$ para $concat(p, q)$.
Para construir esse termo: primeiro, construa a função

$$f(x) : \Pi_{y:A}(x = y) \to \Pi_{z:A}(y = z) \to (x = z)$$

para qualquer $x : A$. Para isso, usando o princípio de indução para tipos identidade, é suficiente construir a função

$$f(x, x, refl_x) : \Pi_{z:A}(x = z) \to (x = z)$$

Essa função é a mesma coisa que $\lambda z.id_{x=z}$ com $id_{x=z} : x = z$. Dessa forma:

$$f(x) := ind - eq_x(\lambda z.id) : \Pi_{y:A}(x = y) \to \Pi_{z:A}(y = z) \to (x = z)$$

Por fim, seja $p : x = y$ e $q : y = z$, tem-se que:

$$concat_{x,y,z}(p, q) := f(x, y, p, z, q)$$

Definição 9.16 ((RIJKE, 2022))**.** Seja A um tipo. A **operação de inversão** é:

$$inv : \Pi_{x,y:A}(x = y) \to (y = x)$$

Na maioria das vezes, se escreve p^{-1} para $inv(p)$
Pelo princípio da indução para tipos indutivos, para construir a inversão é suficiente construir

$$inv(refl_x) : x = x$$

para qualquer $x : A$. Nesse caso $inv(refl_x) := refl_x$
Uma pergunta interessante que surge da concatenação é se ela é associativa, ou seja se é possível identificar

$$(p \centerdot q) \centerdot r \text{ e } p \centerdot (q \centerdot r)$$

para quaisquer $p : x = y$, $q : y = z$ e $r : z = w$ de tipo A.
Na teoria dos tipos, isso é o mesmo que perguntar se existe um termo

$$assoc(p, q, r) : (p \centerdot q) \centerdot r = p \centerdot (q \centerdot r)$$

Pelo princípio da indução para tipos indutivos, é suficiente mostrar que

$$Pi_{z:A}\Pi_{q:x=z}Pi_{w:A}\Pi_{r:z=w}(refl_x \centerdot q) \centerdot r = refl_x \centerdot (q \centerdot r)$$

Pela regra da computação do tipo identidade, $\mathrm{refl}_x \cdot q = q$, ou seja:

$$(\mathrm{refl}_x \cdot q) \cdot r = q \cdot r$$

De forma similar tem-se que $\mathrm{refl}_x \cdot (q \cdot r) = q \cdot r$ e os dois lados são iguais. Ou seja: $\mathrm{assoc}(\mathrm{refl}_x, q, r) := \mathrm{refl}_{q \cdot r}$

Definição 9.17 ((RIJKE, 2022)). Seja A um tipo. É possível definir operações unitárias à esquerda e a direita que atribuem a cada $p : x = y$ as identificações:

$$\mathrm{left} - \mathrm{unit}(p) : \mathrm{refl}_x \cdot p = p$$

$$\mathrm{right} - \mathrm{unit}(p) : p \cdot \mathrm{refl}_x = p$$

respectivamente.

Pela eliminação da identificação, é suficiente construir

$$\mathrm{left} - \mathrm{unit}(\mathrm{refl}_x) : \mathrm{refl}_x \cdot \mathrm{refl}_x = \mathrm{refl}_x$$

$$\mathrm{right} - \mathrm{unit}(\mathrm{refl}_x) : \mathrm{refl}_x \cdot \mathrm{refl}_x = \mathrm{refl}_x$$

Em ambos os casos tem-se $\mathrm{refl}_{\mathrm{refl}_x}$

Definição 9.18 ((RIJKE, 2022)). Seja A um tipo, é possível definir operações de inversão à esquerda e à direita da seguinte forma:

$$\mathrm{left} - \mathrm{inv}(p) : p^{-1} \cdot p = \mathrm{refl}_y$$

$$\mathrm{right} - \mathrm{inv}(p) : p \cdot p^{-1} = \mathrm{refl}_x$$

Pela eliminação da identificação, é suficiente construir

$$\mathrm{left} - \mathrm{inv}(\mathrm{refl}_x) : \mathrm{refl}_x^{-1} \cdot \mathrm{refl}_x = \mathrm{refl}_y$$

$$\mathrm{right} - \mathrm{inv}(\mathrm{refl}_x) : \mathrm{refl}_x \cdot \mathrm{refl}_x^{-1} = \mathrm{refl}_x$$

Usando a regra da computação:

$$\mathrm{refl}_x^{-1} \cdot \mathrm{refl}_x = \mathrm{refl}_x \cdot \mathrm{refl}_x = \mathrm{refl}_x$$

dessa forma

$$\mathrm{left} - \mathrm{inv}(\mathrm{refl}_x) := \mathrm{refl}_{\mathrm{refl}_x}$$

De forma similar:

$$\mathrm{refl}_x \cdot \mathrm{refl}_x^{-1} = \mathrm{refl}_x \cdot \mathrm{refl}_x = \mathrm{refl}_x$$

e

$$\mathrm{right} - \mathrm{inv}(\mathrm{refl}_x) := \mathrm{refl}_{\mathrm{refl}_x}$$

9.3.2 identificação em funções

Usando o princípio de indução do tipo identidade é possível mostrar que toda função preserva identificações:

Definição 9.19 ((RIJKE, 2022))**.** Seja $f : A \to B$ uma função. A **ação em caminhos** de f é definida como a operação:

$$ap_f : \Pi_{(x,y:A)}(x = y) \to (f(x) = f(y))$$

Além disso, existem operações

$$ap - id_A : \Pi_{(x,y:A)}\Pi_{(p:x=y)}p = ap_{id_A}(p)$$

$$ap - comp(f, g)\Pi_{(x,y:A)}\Pi_{(p:x=y)}ap_g(ap_f(p)) = ap_{g\circ f}(p)$$

Para fazer a construção é necessário mostrar que cada uma dessas vale para a reflexividade. No primeiro caso,

$$ap_f(refl_x) := refl_{f(x)}$$

Já para $ap - id_A$:

$$ap - id_A(refl_x) := refl_{refl_x}$$

E para $ap - comp(f, g)$:

$$ap - comp(f, g, refl_x) := refl_{refl_{g(f(x))}}$$

Definição 9.20 ((RIJKE, 2022))**.** Seja $f : A \to B$ uma função. Existem identificações:

$$ap - refl(f, x) : ap_f(refl_x) = refl_{f(x)}$$

$$ap - inv(f, p) : ap_f(p^{-1}) = ap_f(p)^{-1}$$

$$ap - concat(f, p, q) : ap_f(p \cdot q) = ap_f(p) \cdot ap_f(q)$$

para todo $p : x = y$ e $q : x = y$

Para construir $ap - refl(f, x)$ é simples ver que $ap_f(refl_x) = refl_{f(x)}$, então:

$$ap - refl(f, x) := refl_{refl_{f(x)}}$$

Para construir $ap - inv(f, p)$, usando a reflexividade como p, fica como:

$$ap - inv(f, refl_x) := refl_{ap_f(refl_x)}$$

Para consturir $ap - concat(f, p, q)$ é necessário somente usar a eliminação em p:

$$ap - concat(f, refl_x, q) := refl_{ap_f(q)}$$

9.3.3 Transport

Dada uma identificação $p : x = y$ em um tipo base A, é possível transportar qualquer elemento $b : B(x)$ para a fibra $B(y)$

Definição 9.21 ((RIJKE, 2022))**.** Seja A um tipo e seja B uma família de tipos sobre A. A operação de **transporte** seja construida como:

$$tr_B : \Pi_{x,y:A}(x = y) \to (B(x) \to B(y))$$

Para construir tr_B é usada indução sobre $p : x =_A y$, tomando

$$tr_B(refl_x) := id_{B(x)}$$

Logo, é possível ver que a teoria dos tipos de Martin-Lof não distingue entre elementos identificados x e y, pois para qualquer família de tipos B sobre A é possível obter um elemento de $B(y)$ dos elementos de $B(x)$. Através do transporte é possível desenvolver uma ação entre caminhos em cima de uma função dependente:

Definição 9.22 ((RIJKE, 2022))**.** Dada uma função dependete $f : \Pi_{a:A} B(a)$ e uma identificação $p : x =_A y$, é possível construir:

$$apd_f(p) : tr_B(p, f(x)) = f(y)$$

A identificação $apd_f(p)$ é construida pelo princípio indutivo do tipo identidade de forma:

$$apd_f(refl_x) : tr_B(refl_x, f(x)) = f(x)$$

e

$$apd_f(refl_x) := refl_{f(x)}$$

9.3.4 unicidade de refl

O tipo identidade é uma *família* indutiva de tipos. $refl$ só é único em relação ao elemento que se quer identificar. Dessa forma, somente o par $(a, refl_a)$ pode ser único no tipo de todos os pares

$$(x, p) : \Sigma_{x;A}\, a = x$$

Proposição 9.2 ((RIJKE, 2022))**.** Considere um elemento $a : A$. Então existe uma identificação

$$(a, refl_a) = y$$

no tipo $\Sigma_{x;A}\, a = x$ para qualquer $y : \Sigma_{x;A}\, a = x$

Prova: pelo princípio indutivo de Σ, é suficiente mostrar que existe uma identificação

$$(a, refl_a) = (x, p)$$

para qualquer $x : A$ e $p : a = x$. Pelo princípio indutivo do tipo identidade é suficiente mostrar, por sua vez, que

$$(a, refl_a) = (a, refl_a)$$

Essa identificação é obtida por reflexividade □

Part V
Semântica Categorial das teoria homotópicas de tipos

Part VI
Lógica

Part VII
Apêndices

10 Apêndice Histórico

Esse apêndice histórico serve como uma forma do leitor se localizar nos fatos mais importantes para as áreas do livro. Cada fato vêm com um parentese antes para definir para qual área, ou áreas, o fato é relevante

- 1958 - (Teoria dos Tipos) publicação por Kurt Gödel do artigo "Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes" (Sobre uma extensão do ponto de vista finitário ainda não utilizada) que representa o início do assim chamado "Sistema T".
- 1984 - (HoTT, ∞-cats) - Publicação por Grothendieck do seu *Esquisse d'un Programme* (Esboço de um programa) onde ele publica a **Hipótese da Homotopia** de que os n-tipos homotópicos podem ser equivalentes ao n-grupoide, com $n \in \mathbb{N} \cup \{\infty\}$
- 1991 - (HoTT, ∞-cats) - Kapranov e Voevodsky publicam uma prova de que a categoria homotópica dos espaços é equivalente à categoria homotópica dos ∞-grupoides fracos (Hipótese da homotopía para $n = \infty$). Essa prova possuia uma falha e essa falha foi a motivação para que Voevodsky desenvolvesse a Homotopy Type Theory
- 1998 - (∞-cats) - Carlos Simpson publica um artigo que possua um contraexemplo ao resultado principal do artigo de 1991 de Kapranov-Voevodsky

References

AWODEY, S. **Category theory**. Oxford: OUP Oxford, 2010.

CHURCH, A. A formulation of the simple theory of types. **Journal of Symbolic Logic**, v. 5, n. 2, p. 56–68, 1940.

GIRARD, J.-Y.; TAYLOR, P.; LAFONT, Y. **Proofs and types**. Cambridge: Cambridge university press, 1989. v. 7.

GÖDEL, K. On a hitherto unexploited extension of the finitary standpoint. **Journal of philosophical logic**, Springer, v. 9, p. 133–142, 1980.

JACOBS, B. **Categorical logic and type theory**. Amsterdam: Elsevier, 1999.

NEDERPELT, R.; GEUVERS, H. **Type theory and formal proof: an introduction**. Cambridge: Cambridge University Press, 2014.

RIEHL, E. **Category theory in context**. Nova Iorque: Courier Dover Publications, 2017.

RIJKE, E. **Introduction to Homotopy Type Theory**. [S.l.: s.n.], 2022.

ROSIAK, D. **Sheaf theory through examples**. Cambridge: MIT Press, 2022.

TAIT, W. W. Intensional interpretations of functionals of finite type i. **The journal of symbolic logic**, Cambridge University Press, v. 32, n. 2, p. 198–212, 1967.