Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sobre a Confluência do Cálculo λ_x com Locally Nameless Representation

Gabriel N. R. Fonseca

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Flávio L. C. de Moura

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Sobre a Confluência do Cálculo λ_x com Locally Nameless Representation

Gabriel N. R. Fonseca

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Flávio L. C. de Moura (Orientador)
CIC/UnB

Prof. Dr. Daniel L. Ventura
INF/UFG

Prof.a Dr.a Daniele Nantes
MAT/UnB

Prof. Dr. Marcelo G. Mandelli
Coordenador do Bacharelado em Ciência da Computação

Brasília, 17 de novembro de 2021

Dedicatória

Dedico à aqueles que me apoiaram, me ensinaram, me corrigiram e me fizeram sorrir. Dedico ao meu pai, João, e à minha mãe, Francisca por todo os dias me incentivarem a sonhar mais alto. Dedico também à meus irmãos, João Gabriel e Cecília, pela alegria que vocês me proporcionam. Por fim, dedico aos meus amigos, que sempre estiveram por mim.

Agradecimentos

Agradeço ao corpo docente da Universidade de Brasília (UnB) por todo o aprendizado ao longo dos anos de curso. Especialmente, agradeço à Professora Carla Castanho e meu orientador Flávio Moura, pela excelente conduta para com os alunos, sempre dispostos a elevar o nível do aprendizado.

Agradeço também à instituição UnB pela representatividade na luta por democracia e igualdade social. Por fim, agradeço à todo o povo dessa nação que financia meu estudo nesta instituição de ensino.

Resumo

O Calculo Lambda com Substituições Explícitas é uma expansão da gramática do Cálculo Lambda proposto por A. Church na década de 1930. Múltiplas implementações desse cálculo existem, umas delas é denominada λ_x. Como demonstrado por K. Nakazawa, o sistema λ_x é confluente. A demonstração ocorre pela construção de uma relação entre este sistema de reduções e a propriedade Z. Este trabalho apresenta uma tentativa da formalização dessa prova via assistente de provas Coq.

O cálculo λ_x é implementado neste trabalho expandindo o *framework locally nameless representation*. Este *framework* foi introduzido por A. Charguéraud e traz uma noção mista entre a representação com nomes e com Índices de De Bruijn. Ao longo do trabalho é exposto que a formalização não pode ser concluída por estes meios pois, a noção de substituição para as substituições explícitas do cálculo λ_x difere da noção de substituições da meta-substituição proposta pelo *framework*. Além disso, esta meta-substituição fere o Lema da Substituição em seu caso geral. Trabalhos futuros incluem a reimplementação da meta-substituição e a conclusão da formalização via assistentes de prova.

Palavras-chave: calculo, lambda, substituições, explicitas, de bruijn, coq, locally, nameless

Abstract

The Lambda Calculus with Explicit Substitutions is a grammar's expansion of the Lambda Calculus propounded by A. Church in the 1930s. Multiple implementations of this calculus exist; one of them is named λ_x. As demonstrated by K. Nakazawa, the λ_x is a confluent system. The demonstration is done by relating this reductions system and the Z property. This work explores this formalization through Coq proof assistant.

In this work, the λ_x calculus is implemented by an expansion of the locally nameless representation framework. Introduced by A. Charguéraud, it brings a mixed notion between the representation with names and De Bruijn Indexes. Through this work, it is made clear that the formalization can't be concluded by these means because the λ_x calculus notion of substitutions differs from the notion for the meta-substitution propounded by the framework. Besides that, the meta-substitution does not apply to the general case of the Substitution Lemma. The next steps include the reimplementation of the meta-substitution and the conclusion of the formalization via proof assistants.

Keywords: calculus, lambda, substitutions, explicit, de bruijn, coq, locally, nameless

Sumário

Lista de Tabelas

Capítulo 1

Introdução

A tarefa de desenvolver software confiável e seguro é muito difícil e é objeto de estudo dentro da ciência da computação. A crescente escala e complexidade dos sistemas modernos, aliado à quantidade de demandas que os mesmos atendem e à quantidade de pessoas envolvidas elevam o esforço necessário para garantir-se correção no produto final. Em razão disso, cientistas da computação desenvolveram técnicas para atender de maneira satisfatória tal interesse. São essas as técnicas de gerenciamento de projetos (e.g. Scrum, DevOps), padrões de projeto (e.g., MVC, Plugin), técnicas de programação (e.g., programação orientada à objetos, programação funcional) e também técnicas matemáticas para validar propriedades a respeito dos programas [1].

Para a maior parte das aplicações, em especial aquelas orientadas a um negócio, as três primeiras técnicas cumprem bem a sua função de garantir o bom funcionamento do software, atestado pela manutenibilidade do produto e pela satisfação geral do usuário final. O custo de adoção das mesmas é relativamente baixo por uma série de fatores, como a rápida curva de aprendizagem, disponibilidade massiva de *frameworks* bem estudados e estabelecidos, velocidade o qual as mesmas podem ser implementadas, dentre outros. Contudo, ao olhar para o conjunto de softwares em que a plena correção importa, tais técnicas não são suficientes para garantir que estes softwares apresentem as propriedades que deles se esperam. Nesse âmbito, requer-se a formalização matemática como via para indubitavelmente validar as proposições sobre os mesmos.

Os programas os quais a verificação formal é de interesse são aqueles ditos de missão crítica. Tais quais a indisponibilidade, perda de dados, erros de cálculo ou falhas de processamentos trazem não só prejuízos financeiros bem como sociais ou que podem representar a quebra de uma organização. Como por exemplo: controladores de voo, *firmware* de processadores, aplicativos online de bancos. Garantir que tais programas funcionam passa por verificar matematicamente que as partes críticas dos mesmos atendem as especificações e que as mesmas interagem da forma esperada com as demais. Isto

requer certa maestria em relação a provas matemáticas.

As provas matemáticas podem ser resumidas a uma argumentação lógica e estruturada de que uma proposição (ou propriedade) vale. O autor da prova utiliza de um sistema lógico e de um sistema dedutivo com regras pré-estabelecidas para escrever a argumentação a respeito do porquê uma proposição é válida. Todavia, até mesmo os matemáticos mais habilidosos podem cometer erros em suas argumentações, levando a inconsistências em suas conclusões. Para garantir a correção, existem os assistentes de prova.

É sabido que a matemática alimenta a ciência da computação, até mesmo pode-se dizer que esta é uma área de matemática. Contudo, essa relação não é unidirecional, tendo a ciência da computação contribuído para o campo da lógica com a construção de ferramentas que auxiliam em provas matemáticas. São estes os provadores automáticos de teoremas (PAT) e os assistentes de prova. Os PATs têm aplicação limitada, recebem como entrada uma proposição e por meio de alguma heurística tentam decidir sobre a mesma dentro de um limite tempo. Já os assistentes de prova automatizam os passos triviais enquanto necessitam de um operador humano para os demais passos. Nesta linha, os assistentes têm um papel muito mais relevante na verificação de provas (ou pelo menos de uma ideia) do que na construção de uma prova do zero.

As provas via assistente ou PAT costumam ter credibilidade maior do que aquelas feitas via argumentação em papel e lápis. Todo o passo-a-passo é verificado frente ao sistema dedutivo empregado, ou seja, não há saltos lógicos tampouco aplicações incorretas de regras. Exemplos de aplicações que tiveram suporte de assistentes de prova são o CompCert - Compilador Otimizado para C [2] e componentes da arquitetura de processadores RISC-V [3]. Além disso, em um âmbito mais teórico, a própria formalização do Teorema das Quatro Cores por G. Gonthier [4] e da Conjectura de Kepler por T. Hales [5] foram feitas via assistentes de prova.

Um dos assistentes de prova populares é o Coq. Lançado em 1989, este utiliza-se do Cálculo de Construções Indutivas para verificar o passo-a-passo de uma prova [6]. Este assistente é popular pois provê uma interface bastante amigável para o desenvolvimento de aplicações certificadas.

Em trabalho desenvolvido por K. Nakazawa, o autor formaliza a propriedade da confluência para uma série de extensões do Cálculo Lambda [7]. Dentre elas o Calculo Lambda com Substituições Explícitas λ_x. O Cálculo Lambda com Substituições Explícitas descreve uma expansão da gramática do Cálculo Lambda proposto por A. Church que busca aproximar modelo teórico e implementação [8][9][10]. O objetivo desse trabalho é verificar via Coq o Lema da Confluência do cálculo λ_x proposto pelo autor. Além disso, apresentar os principais desafios encontrados bem como aqueles inerentes ao processo.

Este documento daqui em diante organiza-se da seguinte forma: O Capítulo 2 expõe

a fundamentação teórica necessária para a compreensão deste trabalho, desde dedução natural, cálculo-lambda e a variante com substituições explicitas, notação de De Bruijn, até uma seção sobre o assistente de provas Coq. O capítulo 3 aborda o desenvolvimento do trabalho em conjunto com as discussões a respeito dos principais lemas. O capítulo 4 retoma as discussões a respeito do desenvolvimento agora com enfoque nos pontos em aberto e nos desafios encontrados. Por fim, o capítulo 5 conclui o trabalho repassando por tudo aquilo discutido anteriormente. As demonstrações relativas à confluência do cálculo λ_x, via assistente de provas Coq, podem ser encontradas em https://github.com/nunesgrf/lx-confluence.

Capítulo 2

Fundamentação Teórica

Este capítulo abordará os conceitos e ferramentas necessárias para a compreensão integra desde trabalho. A ideia é que as fundações para as discussões das seções posteriores sejam consolidadas e que também o leitor tenha um ponto de partida para se aprofundar nos tópicos aqui abordados. Este capítulo terá caráter introdutórios aos temas.

O capítulo divide-se em uma seção de fundamentação sobre Dedução Natural. Segue-se por um seção sobre Cálculo Lambda com enfoque em Substituições Explícitas bem como Índices de De Bruijn. Finalizando o capítulo, uma seção sobre o assistente de provas Coq, buscando apresentar o uso do mesmo em uma demonstração simples.

2.1 Dedução Natural

Desenvolvido por G. Gentzen na década de 1930, dedução natural é um sistema dedutivo que tenta se assemelhar ao raciocínio natural sobre proposições [11]. A partir de um conjunto de fórmulas infere-se uma nova fórmula resultante seguindo um conjunto de regras de inferência [12]. Esse processo pode ser expresso em uma árvore de dedução. Cada transição nos nós representa um sequente; dividido entre antecedente, a informação já obtida, e consequente, o que se conclui à partir do antecedente. O conjunto de regras de inferências variam conforme o sistema dedutivo.

$$\frac{antecedente_1 \quad antecedente_2 \quad ... \quad antecedente_n}{consequente}\,(\text{regra de inferência})$$

Os sistemas dedutivos possuem gramáticas próprias. A gramática define o que são termos e fórmulas à serem operados dentro do sistema dedutivo. Em dedução natural, a gramática é composta por termos, formulas definidas recursivamente em meio a operações

Termos	$t := x \mid f(t, \ldots, t)$
Fórmulas	$f := p(t, \ldots, t) \mid \bot \mid f \wedge f \mid f \to f \mid f \vee f \mid \neg f \mid \exists_x f \mid \forall_x f$

Tabela 2.1: Fórmulas e termos da Lógica de Primeira Ordem

e quantificadores. Esta gramática pode ser visualizada na seguinte Tabela 2.1.

Uma regra de inferência determina que dado um conjunto de fórmulas como hipótese, outro conjunto de fórmulas pode ser inferido. Qualquer interpretação que satisfaz a hipótese também satisfaz a conclusão. A aplicação encadeada das regras é o que forma a derivação ou prova. A Tabela 2.2 detalha o conjunto de regras de inferência para dedução natural.

Enunciando um lema com a notação $A_1, A_2, ..., A_n \vdash C$, as n premissas ficam ao lado esquerdo de $\vdash$ e a proposição inferida ao lado direito. Para construir a demonstração de que o lema vale, é necessário construir a árvore de dedução valendo-se somente das regras da tabela. Além disso, todas as hipóteses criadas nos ramos devem ser descartadas utilizando algumas das regras que realize descarte. Ferir algum desses princípios torna a prova inconsistente e não permite concluir a correção do lema. A seguir exemplos de construções aplicando o sistema lógico:

Exemplo. $A \to B, B \to C \vdash A \to C$

$$\dfrac{\dfrac{B \to C \qquad \dfrac{[A]^a \quad A \to B}{B}\,(\to_e)}{C}\,(\to_e)}{A \to C}\,(\to_i)_a$$

Exemplo. $\neg A \vee B, A \vdash B$

$$\dfrac{\neg A \vee B \quad [B]^b \qquad \dfrac{\dfrac{[\neg A]^a \quad A}{\bot}\,(\bot_e)}{B}\,(\vee_e)_{b,a}}{B}$$

Exemplo. $\neg A \vee B \vdash A \to B$

$$\dfrac{\dfrac{\neg A \vee B \quad [B]^b \qquad \dfrac{\dfrac{[\neg A]^c \quad [A]^a}{\bot}\,(\bot_e)}{B}\,(\vee_e)_{b,c}}{B}}{A \to B}\,(\to_i)_a$$

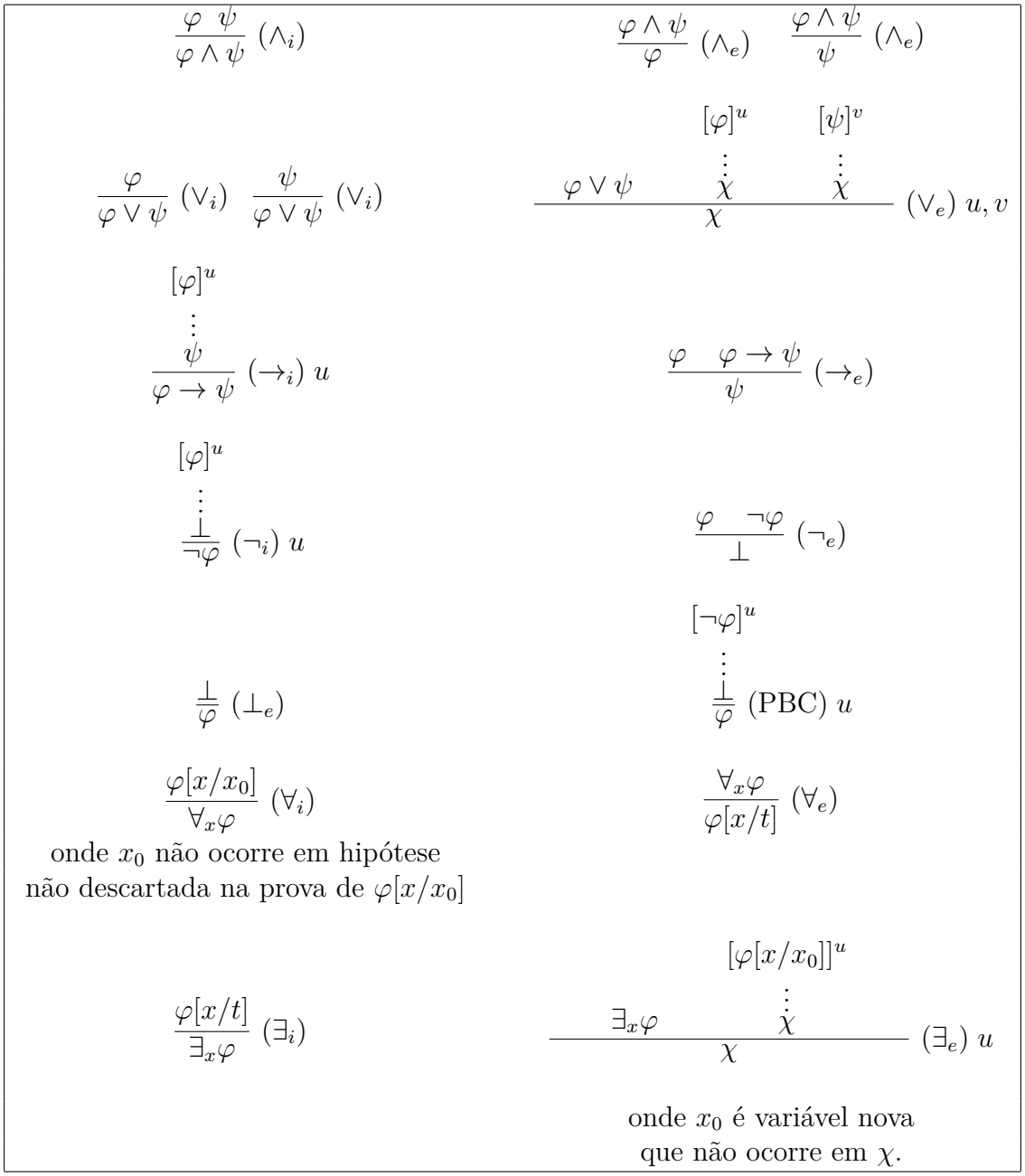


Tabela 2.2: Regras de Inferência para Dedução Natural

2.2 Cálculo Lambda

Introduzido na década de 1930 por A. Church, o Cálculo Lambda é a teoria não tipada de funções. Este descreve a noção de função dentro de uma perspectiva computacional. Uma função em Cálculo Lambda é uma caixa preta que recebe um valor como entrada

e retorna uma saída. Como a função opera a entrada não importa para a computação, restrito o fato que esta não possui nenhum estado interno. Há muito pouco definido à priori no Cálculo Lambda.

O Cálculo Lambda possui somente variáveis, um forma de se construir funções e uma forma de aplicá-las. Não há tipos de dados, recursões ou estruturas de controle. Tudo deve ser de alguma forma codificado utilizando as ferramentas que se tem. De fato, é possível codificar qualquer computação utilizando-as. Essa sintaxe pode ser expressa pela seguinte gramática:

Definição 2.1. *Gramática do Cálculo Lambda*

$$M \ := \ x \mid \lambda x.M \mid MM$$

A definição de uma função em Cálculo Lambda lembra bastante a notação usual algébrica. Assim como as funções algébricas, as funções lambda são redefinições do valor de entrada. Para exemplificar isto, suponha uma função $Increment$ que recebe uma entrada x e retorna este valor incrementado em um. Assumindo a definição de operações matemáticas, a codificação dessa função pode ser expressa da seguinte forma:

Exemplo.

$$Increment := \lambda x.x + 1$$

Já uma função Sum que retorna a soma entre dois valores x e y pode ser codificada como:

Exemplo.

$$Sum := \lambda x.\lambda y.x + y$$

Agora supondo uma função *Not* que recebe como entrada um valor booleano - True ou False - e retorna o contrário desde valor. Isso leva a outro tipo de raciocínio importante para a compreensão do cálculo. Cálculo Lambda não possui definição de valores True e False. Portando, é preciso antes de tudo defini-las. É possível entender True como função que decide entre dois valores, sempre escolhe o primeiro em detrimento do segundo. False, por sua vez, sempre escolhe o segundo. Isto leva as seguintes definições:

Definição 2.2. *Valores booleanos no Cálculo Lambda.*

$$True := \lambda x.\lambda y. \ x$$
$$False := \lambda x.\lambda y. \ y$$

Existindo definições de True e False permite-se definir *Not* conforme a especificação. Uma possível definição é:

Definição 2.3. *Operador Not*

$$Not := \lambda x.\ x\ False\ True$$

Percebe-se que *Not* vale-se da definição dos valores booleanos, que na origem também são funções, para a própria construção. Recebendo como entrada um valor booleano x, a função *Not* retorna o valor semanticamente contrário. Uma vez que esse valor de entrada é operado conforme a definição, o resultado esperado é obtido. É possível visualizar isso na seguinte aplicação utilizando *True* como entrada (É fácil verificar que *False* como entrada também produz a saída correta):

Exemplo.

$Not\ True$	*expandindo a definição de Not*
$(\lambda x.\ x\ False\ True)\ True$	*aplicando True*
$True\ False\ True$	*expandindo a definição do True mais à esquerda*
$(\lambda x.\lambda y.\ x)\ False\ True$	*aplicando False e True respectivamente*
$False$	

No cálculo lambda, as variáveis podem ser classificadas entre livres e ligadas. Sempre que uma variável está atrelada a uma abstração esta é dita ligada. As demais são ditas livres. Seguindo esta definição, uma variável por si só é sempre considerada uma variável livre, e em uma aplicação o conjunto das variáveis livres é a união das variáveis livres das duas partes. O conjunto das variáveis livres pode ser definido formalmente pela função $FV(t)$ onde t é um termo lambda:

Definição 2.4. *Conjunto de variáveis livres de t.*

$$FV(t) = \begin{cases} \{t\} & t\ \textit{variável,} \\ FV(M) - \{t\} & t = \lambda x.M, \\ FV(M) \cup FV(N) & t = MN. \end{cases}$$

Exemplo. *variável x.*

$$FV(x) = \{x\} \qquad \textit{trivialmente.}$$

Exemplo. $\lambda y.x\ x\ y$

$$
\begin{aligned}
FV(\lambda y.x\ x\ y) &= FV(x\ x\ y) - \{y\} \\
&= FV(x) \cup FV(x\ y) - \{y\} \\
&= ... \\
&= \{x\}
\end{aligned}
$$

Exemplo. $(\lambda x.\ y)(\lambda x.\ z\ x)$

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

2.2.1 Alfa-conversão

Tendo dois termos lambda $M := \lambda x.\ x$ e $N := \lambda y.\ y$. Tais funções recebem como argumento um valor e retornam-o. Dado que M e N representam a mesma função identidade, a escolha da variável na abstração lambda não realmente importa. Diz-se então que M e N são alfa-equivalentes. Mais concretamente, dois termos são alfa-equivalentes quando estes diferem apenas pelo nome da variável ligada. O processo de conversão entre dois termos alfa-equivalentes é nomeado alfa-conversão.

Apesar de M e N serem alfa-equivalentes, as variáveis x e y por si só não são alfa-equivalentes. Por não estarem ligadas à abstrações lambda, são variáveis livres. Quando a alfa-conversão é aplicada sobre uma abstração, esta somente renomeia as ocorrências de variáveis vinculados a mesma abstração. A alfa-equivalência não existe entre dois termos quando a conversão resulta na captura da variável por outra abstração lambda. A seguir exemplos de termos alfa-equivalentes e não alfa-equivalentes.

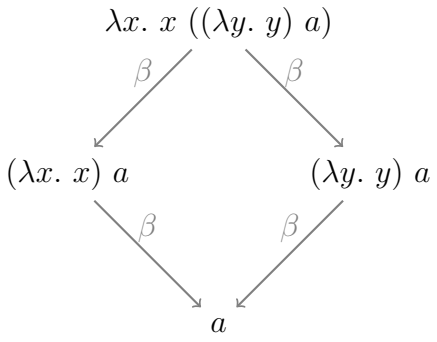
Exemplo. $=_\alpha$ *denota igualdade módulo alfa-conversão, a alfa-equivalência.*

$\lambda x.\ x + 1 =_\alpha \lambda y.\ y + 1$	*Este caso é trivialmente alfa-equivalente.*
$\lambda x.\lambda x.x =_\alpha \lambda y.\lambda x.x$	*Conversão da variável da abstração mais à esquerda.*
$\lambda x.\lambda x.x \neq_\alpha \lambda y.\lambda x.y$	*A conversão resultaria na captura da variável.*
$\lambda x.\lambda y.x \neq_\alpha \lambda y.\lambda y.y$	*Pelo mesmo motivo do exemplo anterior.*

2.2.2 Substituições e Beta-redução

Denotada por $(\lambda x.\ t)s \rightarrow_\beta t[x := s]$, a beta-redução é a operação que reduz a aplicação $(\lambda x.\ t)s$ para o termo $t[x := s]$, por sua vez, $t[x := s]$ representa o termo t onde as ocorrências livres de x são substituídas por s. A beta-redução é essencialmente a operação que corresponde a um passo computacional. Quando não é possível aplicar a beta-redução sobre um termo diz-se que este alcançou a forma normal. É possível a existência de múltiplas beta-reduções para um termo mas no máximo uma forma normal. De fato, essa propriedade é consequência da confluência do Cálculo Lambda.

A propriedade da confluência no Cálculo Lambda enuncia que se existem duas ou mais beta-reduções distintas sobre uma aplicação, então existe um termo comum nas cadeias de reduções. Isto garante que qualquer que seja a heurística que determina qual a próxima redução dentre as possíveis, o resultado sempre será o mesmo. O exemplo abaixo mostra aplicação dupla de funções identidade com a variável a e como a escolha do caminho das reduções é indiferente para o resultado final.



À primeira vista é razoável concluir que sempre a forma normal fará parte da intersecção entre dois caminhos de beta-reduções. A forma normal é o termo o qual não há mais reduções possíveis e para onde todos os caminhos deveriam convergir. Contudo, nem sempre é possível alcançar a forma normal. Isto ocorre quando os passos de redução levam à um termo previamente alcançado. Gerando assim um laço infinito de beta-reduções. O termo $\Omega := (\lambda x.\ xx)(\lambda x.\ xx)$ é um exemplo de aplicação que não pode ser reduzida a uma forma normal. Resolvendo a aplicação, o resultado é o próprio Ω. Por si só, este termo não tem grandes utilidades, mas este, associado a outras construções, codifica a noção de recursão no Cálculo Lambda.

O conceito de substituição implícito até então neste documento é que as substituições funcionam como uma busca textual. As substituições performadas até o momento foram simples o suficiente para que esta heurística de substituição funcionasse. Todavia, existem uma coleção de casos onde esta falharia. Considere as seguintes reduções:

Exemplo. $(\lambda a.\lambda a.\ a)(\lambda b.\ b) \rightarrow_\beta (\lambda a.\ a)[a := \lambda b.\ b]$

Se a substituição é aplicada textualmente sobre o termo à ser reduzido, então o resultado seria $\lambda a.\lambda b.\ b$. O que está efetivamente incorreto. Basta interpretar que este termo é na verdade uma função que recebe um argumento, ignora-o, e retorna uma função identidade. O modo como a substituição é feita deve levar em conta a qual abstração cada variável está ligada.

Exemplo. $(\lambda z.\lambda y.\ z)(\lambda x.\ y) \rightarrow_\beta (\lambda y.\ z)[y := \lambda x.\ y]$

Similar ao caso anterior, uma substituição direta levaria resultaria em uma interpretação incorreta da regra. Substituindo z por $\lambda x.\ y$ resultaria na variável y capturada pela abstração. Há uma colisão de nomes entre os termos, antes de aplicar a regra o termo $\lambda y.\ z$ deve passar por uma alfa-conversão. Este tipo de raciocínio deve estar sistematizado nas regras de substituição.

A função de substituição deve substituir todas as ocorrências livres em um termo. A definição desta função deve ser feita analisando a estrutura do mesmo. Desta forma, é possível construir uma definição que evita a captura de variáveis e que vale-se do conceito de alfa-equivalência. Especialmente quando o termo o qual se aplica a substituição é uma abstração. Isto leva a seguinte definição:

Definição 2.5. *Regras de substituição do Cálculo Lambda*

- $x[x := E] = E$
- $y[x := E] = y,\ y \neq x$
- $(MN)[x := E] = M[x := E]\ N[x := E]$
- $(\lambda x.M)[x := E] = \lambda x.M$
- $(\lambda y.M)[x := E] = \lambda y.(M[x := E]), y \neq x\ e\ y \notin FV(E)$

Caso seja o termo seja uma variável, a resolução é direta, dependendo apenas da igualdade entre esta variável e o termo a ser substituído. No caso da aplicação, apenas propaga-se as substituições sobre ambas as partes. A abstração, por sua vez, divide-se em duas partes: O termo a ser substituído ser a variável ligada à abstração e quando este é possivelmente uma variável livre no corpo desta abstração.

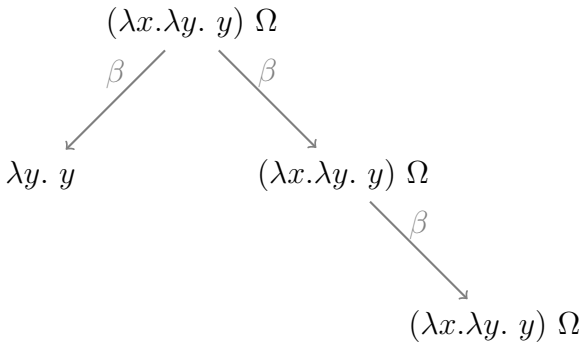
No caso da substituição da variável ligada, a substituição cessa. Esta não ocorre livremente dentro do corpo da abstração. Logo, não há substituições a serem aplicadas. No caso da variável livre, é necessário que a variável ligada ao corpo da abstração não seja livre no termo introduzido pela substituição. Caso venha a ocorrer, basta aplicar a substituição em um termo alfa-equivalente.

2.2.3 Teorema de Church-Rosser

O Calculo Lambda pode ser lido como um sistema de reduções abstrato. Os passos de computação do cálculo são beta-reduções. A noção de confluência existe então para o Cálculo Lambda. De fato, o Teorema de Church-Rosser é a demonstração da confluência do cálculo[13]. Por consequência direta, se um termo possui uma forma normal, então esta é única.

Ao reduzir termos lambda, essencialmente, são possíveis duas estratégias de redução. A primeira, por valor, prioriza por reduzir os argumentos da aplicação. Já a segunda, por nome, busca protelar ao máximo a redução dos argumentos. Tais tipos de reduções podem parecer abstratas demais mas, a redução por nome é utilizada para implementar o conceito de *laziness* em linguagens de programação como *Haskell* [14]. Isto evidencia um pouco da relevância do estudo sobre a Confluência no Cálculo Lambda.

A estratégia de redução por nome mais permissiva do que a por valor [15]. Reduções por valor podem gerar laços no processo de redução. Se um laço ocorre, ainda existem reduções pendentes porém, não há como obter a forma normal por aquela estratégia. Isto pode ser visto no seguinte exemplo com operador $\Omega = (\lambda x.xx)(\lambda x.xx)$.



O ramo mais a esquerda reduz para a forma normal em um passo utilizando a redução por nome. Para esta estratégia, não importa o valor de Ω até este seja necessário. O argumento Ω é descartado pela abstração lambda. Resultando na função identidade. O ramo mais a direita pode ser reduzido indefinidamente. Como exposto na seção 2.2.2, o operador Ω reduz para si próprio. Neste caso, a cadeia de reduções por valor jamais termina.

2.2.4 Substituições Explícitas

As substituições em Cálculo-Lambda são geradas pela beta-redução. A regra beta é responsável por disparar o processo de substituição. Apesar de a substituição ser bem definida, o processo da beta é definido informalmente e ocorre no meta-cálculo. Isto leva a

alguns problemas práticos, a correspondência entre teoria e implementação torna-se não-trivial. Dado que as substituições não fazem parte da gramática do cálculo, a operação é teoricamente feita de forma automática. Tornando uma implementação fiel e correta do cálculo muito complexa. Em razão disso, inclui-se formalmente as substituições no Cálculo Lambda.

O Cálculo Lambda com Substituições Explicitas é uma expansão da gramática do Cálculo Lambda original. As substituições deixam de fazer parte do meta-cálculo e agora estão explicitamente codificadas na gramática. Isto leva a algumas mudanças em como estas são operadas. Sendo assim, a gramática original do Cálculo Lambda é expandida com um novo construtor:

Definição 2.6. *Gramática do Cálculo Lambda com Substituições Explícitas*

$$M \ := \ x \mid \lambda x.M \mid MM \mid M\langle x := M\rangle$$

Assim como no Cálculo Lambda original, as substituições (explícitas) necessitam evitar a captura de variáveis mas agora com as substituições sendo um termo. Agora a beta-reduções é feita em diversas etapas. A ideia geral ao definir explicitamente as substituições consiste em ter uma regra que dispara a simulação da beta-redução, seguida de um conjunto de regras que completa essa simulação. Existem diversas formas de concretizar essa simulação. Em outras palavras, existem diversos Cálculo Lambda com Substituições Explícitas [16][17][18][19]. O exemplo mais simples neste contexto é conhecido como λ_x[20]. As regras de substituição deste podem ser definidas como:

Definição 2.7. *Regras de substituição de* λ_x

- $(\lambda x.\ t)N \rightarrow_\beta t\langle x := N\rangle$
- $x\langle x := N\rangle \rightarrow N$
- $y\langle x := N\rangle \rightarrow y,\ y \neq x$
- $(t_1\ t_2)\langle x := N\rangle \rightarrow t_1\langle x := N\rangle\ t_2\langle x := N\rangle$
- $(\lambda y.\ t)\langle x := N\rangle \rightarrow \lambda y.(t\langle x := N\rangle),\ y \neq x\ e\ y \notin FV(N)$

2.2.5 Índices de De Bruijn

Como visto anteriormente, certas abstrações lambda expressam a mesma função mesmo os termos diferindo. São termos alfa-equivalentes. Contudo, com variáveis definidas explicitamente, definir uma implementação de substituições matematicamente correta

torna-se complexo. Existem outras formas de expressar variáveis diferentes da abordagem apresentada até aqui. Uma forma conveniente é utilizando Índices de De Bruijn.

Índice de De Bruijn trata-se de uma notação em que as variáveis de um termo são substituídas por números naturais. Cada número, ou índice, representa a posição relativa da variável para com sua abstração. Desta forma, quaisquer termos que codificam uma mesma função f qualquer serão exatamente idênticos. Esta representação permite à implementação do cálculo desconsiderar a renomeação de variáveis uma vez que termos alfa-equivalentes têm representação única.

As variáveis livres por sua vez, são mantidas por um contexto. O contexto mapeia as variáveis para um índice. Estas são representadas na estrutura dos termos pelo índice associado adicionado pelo nível de abstração a qual a variável livre encontra-se. Alguns exemplos de uso desta representação são:

Nomes	De Bruijn	
$\lambda x.x$	$\lambda 0$	A posição relativa de x para com a abstração é 0.
$\lambda x.\lambda y.y$	$\lambda\lambda 0$	Apenas y tem um índice correspondente.
$\lambda x.\lambda y.x$	$\lambda\lambda 1$	A posição relativa de x para com a abstração é 1.
$(\lambda x.xx)(\lambda x.xx)$	$(\lambda 00)(\lambda 00)$	Os 0s representam variáveis diferentes.
$\lambda x.(\lambda y.xyz)xyz$	$\lambda(\lambda 102)021$	Considerando o contexto $[z, y]$.

Tomando o exemplo com Índices de De Bruijn $\lambda\lambda 1$. Supondo a seguinte substituição $(\lambda\lambda 1)[1 := (0\ 1)]$. Uma substituição puramente textual resultaria em um termo $\lambda\lambda 01$ onde as variáveis livres foram capturadas pelas abstrações. A substituição de forma correta deve evitar isto modificando os índices. Para isso, é necessário definir uma função auxiliar de incremento dos índices:

Definição 2.8. *[21]Função shifting de incremento*

$$\uparrow^i (t) = \begin{cases} \begin{cases} t & t < i, \\ t+1 & t \geq i \end{cases} & t \text{ índice,} \\ \lambda.(\uparrow^{i+1} (M)) & t = \lambda M \\ \uparrow^i (M) \uparrow^i (N) & t = MN \end{cases}$$

O valor i na definição de $\uparrow^i$ é o valor de marcação para quais índices devem ser incrementados naquele nível de abstração. Este valor é inicialmente 0 e é incrementado a medida que encontra-se outras abstrações no corpo. Valendo-se desta definição, é possível definir as regras de substituição:

Definição 2.9. *[21]Substituições com Índices de De Bruijn*

- $i[i := E] = E$
- $j[i := E] = j,\ j < i$
- $j[i := E] = j - 1,\ j > i$
- $(MN)[i := E] = M[i := E]\ N[i := E]$
- $(\lambda M)[i := E] = \lambda M[(i + 1) := \uparrow^0 (E)]$

É digno de nota que ao entrar no corpo de uma abstração, a variável a ser substituída é incrementada em um. Isto tem como objetivo manter a substituição sobre a mesma variável, já que o índice de uma variável é relativo e varia conforme o nível da abstração. Com esta nova definição, é necessária uma beta-redução correspondente. Esta pode ser expressa da seguinte forma:

Definição 2.10. *[21] Beta-redução com Índices de De Bruijn*

$$(\lambda.M)N \rightarrow_\beta \ M[0 := N]$$

Exemplo. $(\lambda\lambda 1)(0\ 1)$

$$\begin{aligned}(\lambda\lambda 1)(0\ 1) &\rightarrow_\beta (\lambda 1)\ [0 := 0\ 1] \\ &= \lambda 1[1 := \uparrow^0 (0\ 1)] \\ &= \lambda 1[1 := 1\ 2] \\ &= \lambda 1\ 2\end{aligned}$$

Exemplo. $(\lambda\lambda 1\ 2)1$

$$\begin{aligned}(\lambda\lambda 1\ 2)\ 1 &\rightarrow_\beta (\lambda 1\ 2)[0 := 1] \\ &= \lambda(1\ 2)[1 :=\uparrow^0 (1)]) \\ &= \lambda(1[1 := 2)\ 2[1 := 2]) \\ &= \lambda 2\ 1\end{aligned}$$

A representação com Índices de De Bruijn adiciona um nível de complexidade maior ao cálculo. Lidar com o incremento dos termos não é uma tarefa simples. Todavia, os índices permitem a adoção de um raciocínio sistemático sobre os termos lambda. Não há mais necessidade de lidar com nomes e compreender as equivalências entre termos. Dado

isto, os índices de De Bruijn são comumente utilizados na implementação e na construção provas em assistentes automatizados. Sendo assim, uma grande adição a este trabalho.

2.3 O Assistente de Provas Coq

Esta seção tem como objetivo fundamentar sobre o básico do assistente de provas Coq. Assistentes de prova podem ser complexos a primeira vista. A ideia é que o leitor conclua a leitura entendendo a dinâmica do assistente, ao menos o suficiente para a compreensão do desenvolvimento do trabalho, e com boas referências caso tenha interesse em aprofundar-se. Uma prova simples em Coq será apresentada e discutida como forma de introduzir o assistente ao leitor leigo.

A prova a ser discutida nesta capítulo é a correção do algoritmo Mergesort. A escolha deste algoritmo se dá ao fato de ser um problema amplamente conhecido, removendo assim uma possível barreira na compreensão da prova. Para fins de simplicidade, será considerada uma versão simplificada do Mergesort que lida apenas com números naturais. Mais à frente o algoritmo será definido formalmente.

O Mergesort recebe como argumento uma lista de números naturais e retorna uma versão ordenada da mesma. O algoritmo realiza esse procedimento valendo-se da técnica divisão-e-coquista. Antes de aferir a correção do algoritmo é preciso definir o que é a correção. Qualquer algoritmo de ordenação é correto se para uma lista l de entrada, o resultado é uma lista ordenada e uma permutação de l [22]. Verificar estas duas propriedades permite concluir a correção do Mergesort.

A construção de uma prova é uma tarefa incremental. Ao longo da prova, é necessário construir pequenas outras provas que combinadas concluem o objetivo maior. Para o caso do Mergesort, tendo a definição do que é a correção e com objetivo de atender as propriedades citadas anteriormente, é necessário definir o que é uma lista ordenada. Uma boa estratégia é definir os construtores de lista e em seguida analisar caso a caso para definir ordenação:

Definição 2.11. *Lista de naturais*

$$l := \begin{cases} nil \\ cons(n, l'), \textit{sendo } n \textit{ natural, e } l' \textit{ lista} \end{cases}$$

Por questão de simplicidade, é utilizada a notação $n :: l$ que representa uma lista com cabeça n e cauda l. Similarmente, $x :: y :: l$ representa uma lista onde x e y são os dois

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

Tabela 2.3: Definição de lista de naturais em Coq.

primeiros elementos da lista e l é a cauda. Já representando uma lista unitária $[x]$, a notação é $x :: nil$. Esta notação torna mais fácil a leitura do que utilizando o construtor $cons(1,\ cons(2,\ cons(3,\ nil)))$ para representar $[1, 2, 3]$, por exemplo.

A partir do construtor de lista é possível definir regras de inferências a respeito da ordenação da lista. É razoável concluir que listas vazias e unitárias são inerentemente ordenadas. Esses são axiomas de base para a definição de ordenação. A partir disto, supondo uma lista $y :: l$ ordenada e x tal que $x \leq y$, então $x :: y :: l$ também é uma lista ordenada. Tais conclusões podem ser expressas via as seguintes regras de inferência:

Definição 2.12. *Regras de inferência sobre ordenação de listas*

$$\frac{}{sorted(nil)}\ (nil\_sorted)$$

$$\frac{}{sorted(n :: nil)}\ (one\_sorted)$$

$$\frac{x \leq y \quad sorted(y :: l)}{sorted(x :: y :: l)}\ (all\_sorted)$$

```
Inductive sorted : list nat → Prop :=
  | nil_sorted : sorted nil
  | one_sorted : ∀ n:nat, sorted (n::nil)
  | all_sorted : ∀ (x y: nat)(l:list nat), sorted(y :: l) → x ≤ y → sor-
ted(x::y::l).
```

Tabela 2.4: Regras de inferência para ordenação de lista de naturais em Coq.

A partir desta primeira definição é possível extrair alguns resultados interessantes para a continuidade da prova. O objetivo é construir as bases para uma prova simples mais a frente. Estes resultados não dizem respeito ao algoritmo mas sim ao predicado de ordenação definido anteriormente.

Os lemas a seguir são provados à partir da indução sobre a estrutura de lista. Consideram-se as formas em que uma lista pode existir segundo a definição. Conforme esta, a lista

pode ser vazia e também um objeto composto por cabeça e cauda, esta por sua vez, outra lista. Se a propriedade vale para ambos os casos, então vale como um todo.

Algumas das árvores de dedução a seguir utilizam propriedades bem conhecidas sobre inteiros ou sobre proposições lógicas por questão de simplicidade da prova. O objetivo é somente apresentar a relação entre as árvores de dedução e a prova em Coq. Mesmo não atendo-se ao formalismo completo, a prova em Coq também está presente. Demonstrando que as propriedades valem.

Lema 2.1. *Se $a :: l$ é uma lista ordenada, então l é uma lista ordenada.*

- $l = nil$

$$\dfrac{\dfrac{[sorted(a :: nil)]^a \quad \dfrac{}{sorted(nil)}\,(\text{nil\_sorted})}{sorted(a :: nil) \wedge sorted(nil)}\,(\wedge_i)}{sorted(nil)}\,(\wedge_e)}{sorted(a :: nil) \rightarrow sorted(nil)}\,(\rightarrow_i)_a$$

- $l = x :: l'$

$$\dfrac{\dfrac{\dfrac{[sorted(a :: x :: l')]^a}{a \leq x \wedge sorted(x :: l')}\,(\text{all\_sorted})}{sorted(x :: l')}\,(\wedge_e)}{sorted(a :: x :: l') \rightarrow sorted(x :: l')}\,(\rightarrow_i)_a$$

```
Lemma tail_sorted: ∀ l a, sorted (a :: l) → sorted l.
Proof.
  intros.
  induction l.
  - apply nil_sorted.
  - inversion H. subst.
    assumption.
Qed.
```

Tabela 2.5: Prova do Lema 2.1 em Coq.

Definição 2.13. $le\_all: \ x \leq_* l := \forall y \in l, \ x \leq y$

Lema 2.2. *Se $a \leq_* l$ e l é uma lista ordenada, então $a :: l$ é uma lista ordenada.*

- $l = nil$

$$\dfrac{[a \leq_* nil \rightarrow sorted(nil)]^a \quad \dfrac{}{sorted(a :: nil)}\,(\text{one\_sorted})}{a \leq_* nil \rightarrow sorted(nil) \rightarrow sorted(a :: nil)}\,(\rightarrow_i)_a$$

- $l = x :: l'$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[a \leq_* x :: l' \wedge sorted(x :: l')]^a}{a \leq_* x :: l'}(\wedge_e)}{a \leq x}(\text{Def. 2.13}) \quad \dfrac{[a \leq_* x :: l' \wedge sorted(x :: l')]^a}{sorted(x :: l')}(\wedge_e)}{a \leq x \wedge sorted(x :: l')}(\wedge_i)}{sorted(a :: x :: l')}(\text{all\_sorted})}{a \leq_* x :: l' \wedge sorted(x :: l') \rightarrow sorted(a :: x :: l')}(\rightarrow_i)_a}{a \leq_* x :: l' \rightarrow sorted(x :: l') \rightarrow sorted(a :: x :: l')}(\text{equivalência})$$

```
Lemma le_all_sorted: ∀ l a, a <=* l → sorted l → sorted (a :: l).
Proof.
  intros.
  induction l.
  - apply one_sorted.
  - apply all_sorted.
    + assumption.
    + unfold le_all in H. apply H. apply in_eq.
Qed.
```

Tabela 2.6: Prova do Lema 2.2 em Coq.

Lema 2.3. *Se $a :: b :: l$ é uma lista ordenada, então $a :: l$ é uma lista ordenada.*

- $l = nil$

$$\dfrac{[sorted(a :: b :: nil)]^a \quad \dfrac{}{sorted(a :: nil)}(\text{one\_sorted})}{sorted(a :: b :: nil) \rightarrow sorted(a :: nil)}(\rightarrow_i)_a$$

- $l = x :: l'$

∇_1:

$$\dfrac{\dfrac{\dfrac{[sorted(a :: b :: x :: l')]^a}{a \leq b \wedge sorted(b :: x :: l')}\,(\text{all\_sorted})}{a \leq b}\,(\wedge_e) \qquad \dfrac{\dfrac{\dfrac{[sorted(a :: b :: x :: l')]^a}{a \leq b \wedge sorted(b :: x :: l')}\,(\text{all\_sorted})}{sorted(b :: x :: l')}\,(\wedge_e)}{b \leq x \wedge sorted(x :: l')}\,(\text{all\_sorted})}{\dfrac{a \leq b \wedge b \leq x}{a \leq x}\,(\wedge_i)}\,(\text{transitividade})$$

∇_2:

$$\dfrac{\dfrac{\dfrac{\dfrac{[sorted(a :: b :: x :: l')]^a}{a \leq b \wedge sorted(b :: x :: l')}\,(\text{all\_sorted})}{sorted(b :: x :: l')}\,(\wedge_e)}{b \leq x \wedge sorted(x :: l')}\,(\text{all\_sorted})}{sorted(x :: l)}\,(\wedge_e)$$

$$\dfrac{\dfrac{\dfrac{\nabla_1 \quad \nabla_2}{a \leq x \wedge sorted(x :: l)}\,(\wedge_i)}{sorted(a :: x :: l')}\,(\text{all\_sorted})}{sorted(a :: b :: x :: l') \rightarrow sorted(a :: x :: l')}\,(\rightarrow_i)_a$$

```
Lemma sublist_sorted: ∀ l a1 a2, sorted (a1 :: a2 :: l) → sorted (a1 :: l).
Proof.
  intros.
  induction l.
  - apply one_sorted.
  - inversion H; subst.
    inversion H2; subst.
    apply all_sorted.
    + assumption.
    + apply Nat.le_trans with a2; assumption.
Qed.
```

Tabela 2.7: Prova do Lema 2.3 em Coq.

Lema 2.4. *Se $a :: l$ é uma lista ordenada, então $a \leq_* l$.*

Diferente dos lemas anteriores, a prova deste lema depende da hipótese de indução. A estratégia utilizada não permite que o objetivo da prova seja construído sem o uso desta hipótese. Para definir a hipótese de indução, basta criar uma regra de inferência que especifique o passo indutivo.

$$\frac{sorted(x :: l')}{x \leq_* l'}\,(\text{H.I.})$$

- $l = nil$

$$\dfrac{\dfrac{[sorted(a :: nil)]^a \qquad \dfrac{\dfrac{[\exists y(y \in a :: nil \wedge a < y)]^b \qquad \bot}{\bot}\,\neg_e}{a \leq_* nil}\,(\text{PBC})_b}{sorted(a :: nil) \rightarrow a \leq_* nil}\,(\rightarrow_i)_a$$

- $l = x :: l'$

$$\dfrac{\dfrac{\dfrac{[sorted(a :: x :: l')]^a}{a \leq x \wedge sorted(x :: l')}\,(\text{all\_sorted}) \qquad \dfrac{\dfrac{[sorted(a :: x :: l')]^a}{a \leq x \wedge sorted(x :: l')}\,(\wedge_e) \qquad \dfrac{sorted(x :: l')}{x \leq_* l'}\,(\text{H.I.})}{a \leq x}\,(\wedge_e)}{a \leq_* x :: l}\,(\text{Def. 2.13})}{sorted(a :: x :: l') \rightarrow a \leq_* x :: l'}\,(\rightarrow_i)_a$$

```
Lemma sorted_le_all: ∀ l a, sorted(a :: l) → a <=* l.
Proof.
  induction l.
  - intros. unfold le_all. intros. simpl In in H0. contradiction.
  - intros. unfold le_all. intros. simpl In in H0. destruct H0 as [H1 |
H2].
    + subst. inversion H; subst. assumption.
    + apply sublist_sorted in H. apply IHl; assumption.
Qed.
```

Tabela 2.8: Prova do Lema 2.4 em Coq.

Definida a ordenação e obtidos alguns resultados interessantes, é preciso também definir o significado de permutação. Esta também é condição necessária para aferir a correção de qualquer algoritmo de ordenação. Permutação de uma lista pode ser expressa como um arranjo dos elementos pertencentes a uma lista. Dadas duas listas l e t, se o número de ocorrências de qualquer elemento em l é igual em t, então l e t são permutações entre si. Isto é, $u_1 := [1, 4, 2, 2]$ é uma permutação de $u_2 = [1, 2, 4, 2]$ pois cada elemento de u_2 aparece exatamente o mesmo número de vezes em u_1. Isto permite definir o predicado de permutação como:

Definição 2.14. *Duas listas l e t são ditas permutações se $\forall n,\ num\_oc(n, l) = num\_oc(n, t)$.*

$$num\_oc(n, l) = \begin{cases} 0 & l = nil, \\ num\_oc(n, l') + 1 & l = a :: l' \ e \ n = a, \\ num\_oc(n, l) & l = a :: l' \ e \ n \neq a \end{cases}$$

```
Fixpoint num_oc n l :=
  match l with
    | nil ⇒ 0
    | h :: tl ⇒
      if n =? h then S(num_oc n tl) else num_oc n tl
  end.
```

Tabela 2.9: Definição em Coq da função de contagem de ocorrências de n em l.

```
Definition perm l l' := ∀ n:nat, num_oc n l = num_oc n l'.
```

Tabela 2.10: Definição de permutação em Coq.

Lema 2.5. *A permutação é reflexiva.*

$$\dfrac{\dfrac{\dfrac{\dfrac{[perm\ l\ l]^a}{num\_oc(n,l) = num\_oc(n,l)}\,(\text{Def. 2.14})}{\top}\,(\text{tautologia})}{(perm\ l\ l) \to \top}\,(\to_i)_a}{perm\ l\ l}\,(\text{tautologia})$$

```
Lemma perm_refl: ∀ l, perm l l.
Proof.
intro l. unfold perm. intro. reflexivity.
Qed.
```

Tabela 2.11: Prova do Lema 2.5 em Coq.

As provas em Coq por vezes divergem da prova em papel e lápis. Isso é devido a expressividade de um método sobre o outro dados os diferentes contextos. Para propriedades simples, a prova em papel e lápis é mais ágil em razão de todo o conhecimento

prévio que o autor carrega em si mesmo. Isto inverte-se quando a prova torna-se complexa e por vezes o autor carrega somente uma ideia da prova e não uma fundamentação completa. Neste quesito, Coq ou qualquer outro assistente de provas permite o autor um maior controle sobre premissas e objetivos da prova. Todas estas dispostas para a visualização do autor.

É importante notar que a prova via árvores de dedução natural dos lemas seguintes seriam demasiadamente grandes e não configuram o objetivo deste trabalho. A base para a compreensão do trabalho pode ser obtida com os exemplos até o momento apresentados. Para uma eventual necessidade de aprofundamento, o leitor pode buscar as referências [12] [22][23]. Dito isso, nos lemas seguintes somente uma ideia sobre a prova e a formalização desta em Coq serão expostos.

Lema 2.6. *Dadas duas listas l_1, l_2 e um n qualquer, onde $l_1 +\!\!+ l_2$ denota a concatenação das duas listas, $num\_oc(n,\ l_1 +\!\!+ l_2) = num\_oc(n,\ l1) + num\_oc(n,\ l2)$.*

A prova desse lema pode ser alcançada aplicando indução sobre a estrutura de l_1. Com isto, dois casos a se provar aparecem: lista vazia e lista não-vazia. O caso da lista vazia é trivialmente verdade. O outro caso depende da análise sobre a igualdade entre a cabeça de l_1 e um n arbitrário. Cada um desses sub-casos gerados também é trivialmente resolvível.

```
Lemma num_oc_append: ∀ n l1 l2, num_oc n l1 + num_oc n l2 = num_oc n
(l1 ++ l2).
Proof.
  intros. induction l1.
  - simpl num_oc. trivial.
  - simpl. destruct (n =? a).
    + rewrite ← IHl1. apply Peano.plus_Sn_m.
    + assumption.
Qed.
```

Tabela 2.12: Prova do Lema 2.6 em Coq.

Definidas as fundações para a prova do algoritmo é necessário de fato definir o que é o algoritmo. Esta etapa em Coq passa por, assim como em papel e lápis, pela escrita do algoritmo. É importante que o algoritmo esteja bem definido pois a definição será usada diretamente na prova. Quando definido um algoritmo recursivo em Coq, o assistente exige a verificação de que a recursão deste algoritmo esteja bem definida: é necessária uma condição de parada e que cada chamada recursiva adicional deve aproxime-se da condição de parada.

Algoritmos de divisão-e-conquista utilizam da estrategia de quebrar o problema original em problemas menores e combinar as soluções. Esta abordagem parte da premissa

que resolver problemas menores é mais simples que atacar o problema inteiro de uma vez. Esta técnica pode ser aplicada em problemas que podem ser quebrados em subproblemas e onde estes não se intersectam [24]. Após isso, os subproblemas devem ser combinados via uma função auxiliar.

O Mergesort divide recursivamente a lista de entrada até o ponto em que os subproblemas sejam listas vazias ou unitárias. Conforme a definição 2.12, as ordenações destas listas são trivialmente resolvíveis. A combinação dos resultados é onde a complexidade do algoritmo acontece. Dadas duas listas ordenadas, ambas são combinadas comparando as cabeças das listas e retornando uma lista construída a partir da menor cabeça e, recursivamente, a combinação do restante. Esse comportamento é implementado pela função MERGE abaixo.

Definição 2.15. *função MERGE auxiliar de combinação de resultados.*

Algorithm 1: MERGE

Data: ρ, η : *Sorted lists*

Result: Sorted merge of ρ and η

```
1 if ρ = nil or η = nil then
2 |   return ρ ++ η
3 else
4 |   if head(ρ) ≤ head(η) then
5 |   |   return head(ρ) :: MERGE(tail(ρ), η)
6 |   else
7 |   |   return head(η) :: MERGE(ρ, tail(η))
  |   _
```

Lema 2.7. *Se p está em $MERGE(l_1,\ l_2)$, então p está em l_1 ou l_2.*

A ideia parte por verificar que o algoritmo em sua saída sempre vai inserir os elementos analisados. É necessário sempre olhar o estado da lista nas chamadas recursivas. A verificação desta propriedade pode ser feita aplicando indução sobre $MERGE(l_1,\ l_2)$. Haverão então três grandes casos a se provar. O primeiro, quando uma das listas é vazia, é trivialmente alcançado. Os demais dependem da hipótese de indução para serem concluídos.

Lema 2.8. *Dadas listas l_1 e l_2, $MERGE(l_1,\ l_2)$ é uma permutação de $l_1 +\!\!+\, l_2$.*

A prova desde lema pode ser alcançada por indução sobre $MERGE(l_1,\ l_2)$. A construção é muito similar ao lema 2.6 exceto que, num primeiro passo, considera-se a implementação de $MERGE$ para a geração dos casos.

```
Lemma merge_in: ∀ y p, In y (merge p) → In y (fst p) ∨ In y (snd p).
Proof.
intros. functional induction (merge p).
  - right. unfold snd. assumption.
    - left. unfold fst. assumption.
    - simpl in H. destruct H as [H1 | H2].
    + left. unfold fst. unfold In. left. assumption.
        + destruct IHl.
        × assumption.
          × left.  unfold fst.  unfold fst in H. simpl In.  right.
assumption.
          × right. simpl. simpl in H. assumption.
    - simpl in H. destruct H as [H1 | H2].
    + right. simpl snd. simpl In. left. assumption.
        + destruct IHl.
        × assumption.
          × left. unfold fst. unfold fst in H. assumption.
          × right. simpl. simpl in H. right. assumption.
Qed.
```

Tabela 2.13: Prova do Lema 2.7 em Coq.

```
Lemma merge_num_oc: ∀ n p, num_oc n (merge p) = num_oc n (fst p) +
num_oc n (snd p).
Proof.
intros. functional induction (merge p).
  - simpl fst. simpl snd. simpl num_oc. trivial.
  - simpl fst. simpl snd. simpl num_oc. trivial.
  - simpl fst. simpl snd. simpl num_oc at 1 2. destruct (n =? hd1).
    + rewrite IHl. apply Peano.plus_Sn_m.
    + rewrite IHl. simpl fst. simpl snd. trivial.
  - simpl fst. simpl snd. simpl num_oc at 1 3. (destruct (n =? hd2)).
      + rewrite IHl. simpl fst. simpl snd. apply Peano.plus_n_Sm.
      + rewrite IHl. simpl fst. simpl snd. trivial.
Qed.
```

Tabela 2.14: Prova do Lema 2.8 em Coq.

Lema 2.9. *Dadas listas l_1 e l_2 ordenadas, $MERGE(l_1,\ l_2)$ é ordenada.*

O objetivo deste lema é verificar se $MERGE$ produz uma lista ordenada em sua saída. Para isso, deve-se utilizar indução sobre $MERGE(l_1,\ l_2)$ e analisar caso a caso. Os casos em que alguma das entradas é uma lista vazia são triviais. Os casos em que as listas são compostas por cabeça e cauda exigem a aplicação dos lemas 2.2, 2.4, 2.7, e da definição 2.13.

A construção é extensa. Apesar de os casos iniciais serem facilmente provados, os dois casos restantes exigem um detalhamento que torna a prova longa. Todavia, mesmo longos, são similares, a compreensão de um deles é suficiente para entender o outro.

A construção pode ser vista separadamente na Tabela 2.18.

Definição 2.16. *Algoritmo Mergesort.*

Algorithm 2: MERGESORT

Data: τ : *List of comparable elements*

Result: *Sorted permutation of* τ

```
if length(τ) ≤ 1 then
    return τ
else
    prefix ← MERGESORT(first_half(τ));
    suffix ← MERGESORT(second_half(τ));
    return MERGE(prefix, suffix);
```

Lema 2.10. *Dada l lista, MERGESORT(l) é ordenada.*

A resolução deste lema é simples. A prova acontece por indução em $MERGESORT(l)$. Para os casos de base, basta aplicar a definição 2.12. Para o caso geral, o mergesort apenas divide recursivamente o problema em problemas menores, a parte de combinação dos resultados é responsabilidade da função $MERGE$. Logo, se $MERGE$ produz uma lista ordenada, então $MERGESORT$ também produz uma lista ordenada. A prova pode ser concluída aplicando o lema 2.9 e a hipótese de indução.

```
Lemma mergesort_sorts: ∀ l, sorted (mergesort l).
Proof.
intro l.
functional induction (mergesort l).
  - apply nil_sorted.
  - apply one_sorted.
  - apply merge_sorts. unfold sorted_pair_lst. split.
    + unfold fst.
      apply IHl0.
    + unfold snd.
      apply IHl1.
Qed.
```

Tabela 2.15: Prova do Lema 2.10 em Coq.

Lema 2.11. *Dada l lista, $MERGESORT(l)$ é permutação de l.*

O mesmo princípio do lema 2.10 aplica-se aqui. Como a responsabilidade da combinação dos resultados está em $MERGE$, basta que $MERGE$ produza uma permutação para que $MERGESORT$ produza uma permutação. A prova pode ser concluída com indução sobre $MERGESORT(l)$. Para os casos-base aplica-se o lema 2.5. Para o caso geral o lema 2.8 e a hipótese de indução.

```
Lemma mergesort_is_perm: ∀ l, perm l (mergesort l).
Proof.
   intro l. functional induction (mergesort l).
   - apply perm_refl.
   - apply perm_refl.
   - unfold perm. intro. rewrite merge_num_oc. unfold fst. unfold snd.
    unfold perm in IHl1. rewrite ← IHl1. unfold perm in IHl0. rewrite
← IHl0.
     rewrite num_oc_append. rewrite firstn_skipn. apply perm_refl.
Qed.
```

Tabela 2.16: Prova do Lema 2.11 em Coq.

Teorema 2.1. *MERGESORT é um algoritmo de ordenação correto*

O algoritmo Mergesort é um algoritmo de ordenação correto. A implementação proposta respeita as propriedades de ordenação e permutação discutidas ao longo dessa seção. Os lemas 2.10 e 2.11 garantem tais propriedades. Dessa forma, a prova pode ser concluída diretamente aplicando-os.

```
Theorem mergesort_is_correct: ∀ l, perm l (mergesort l) ∧ sorted (mergesort
l).
Proof.
intro. split.
  - apply mergesort_is_perm.
  - apply mergesort_sorts.
Qed.
```

Tabela 2.17: Prova do Teorema 2.1 em Coq.

```
Lemma merge_sorts: ∀ p, sorted_pair_lst p → sorted (merge p).
Proof.
  intro p. functional induction (merge p).
  - unfold sorted_pair_lst. intro. destruct H.
    unfold snd in H0. assumption.
  - unfold sorted_pair_lst. intro. destruct H.
    unfold fst in H. assumption.
  - intro. apply le_all_sorted.
    + unfold le_all. intro. intro. apply merge_in in H0.
      destruct H0 as [H1 | H2].
      × simpl fst in H1. unfold sorted_pair_lst in H. destruct H as [H2
H3].
        simpl fst in H2. apply sorted_le_all in H2. unfold le_all in H2.
        apply H2. assumption.
      × simpl snd in H2. apply Nat.le_trans with hd2.
        – apply Nat.leb_le. assumption.
        – unfold sorted_pair_lst in H. destruct H as [H3 H4]. simpl snd
in H4.
          apply sorted_le_all in H4. simpl In in H2. destruct H2 as [H5
| H6].
          ** rewrite H5. trivial.
          ** unfold le_all in H4. apply H4. assumption.
    + apply IHl. unfold sorted_pair_lst. split.
      × simpl fst. unfold sorted_pair_lst in H. destruct H as [H1 H2].
        simpl fst in H1. apply tail_sorted in H1. assumption.
      × simpl snd. unfold sorted_pair_lst in H. destruct H as [H1 H2].
        simpl snd in H2. assumption.
  - intro. apply le_all_sorted.
    + unfold le_all. intro. intro. apply merge_in in H0.
      destruct H0 as [H1 | H2].
      × simpl fst in H1. apply Nat.le_trans with hd1.
        – apply Nat.leb_nle in e0. apply Compare_dec.not_le in e0.
          unfold gt in e0. apply Nat.lt_le_incl. assumption.
        – simpl In in H1. destruct H1 as [H2 | H3].
          ** rewrite H2. trivial.
          ** unfold sorted_pair_lst in H. destruct H as [H4 H5].
             simpl fst in H4. apply sorted_le_all in H4. unfold le_all in
H4.
             apply H4. assumption.
      × simpl snd in H2. unfold sorted_pair_lst in H. destruct H as [H4
H5].
        simpl snd in H5. apply sorted_le_all in H5. unfold le_all in H5.
        apply H5. assumption.
    + apply IHl. unfold sorted_pair_lst. split.
      × simpl fst. unfold sorted_pair_lst in H. destruct H as [H1 H2].
simpl fst in H1.
        assumption.
      × simpl snd. unfold sorted_pair_lst in H. destruct H as [H1 H2].
simpl snd in H2.
        apply tail_sorted in H2. assumption.
Qed.
```

Tabela 2.18: Prova do Lema 2.9 em Coq.

Capítulo 3

A Construção da Prova

Este capítulo tem como objetivo apresentar o Teorema da Confluência para o cálculo λ_x por K. Nakazawa e discutir a estratégia de prova sobre o mesmo, assumindo familiaridade do leitor para com o Cálculo Lambda e Coq no geral. A estratégia de prova passa por reduzir o problema à propriedade Z. O que é a propriedade Z, a relação de Z com a confluência, e como isto pode ser expresso em Coq é o assunto deste capítulo.

3.1 Propriedade Z

Um Sistema de Reduções Abstrato (SRA) é um par ordenado (A, R) composto por um conjunto A e uma relação binária R sobre A. Dados x e $y \in A$, se x reduz para y via R em um passo, isto é $(x, y) \in R$, então é escrito $x \to_R y$. Caso x reduza para y em zero ou mais passos, o par (x, y) faz parte do fecho transitivo-reflexivo de R e pode ser denotado como $x \to_{R*} y$. Qualquer termo reduz para si próprio em zero ou mais passos e se um termo a reduz para b em um passo e b, por sua vez, reduz para c em zero ou mais passo, transitivamente a reduz para c em zero ou mais passos. A partir dessa afirmações, é possível definir regras de inferência:

Definição 3.1. *Dado o SRA (A, R), para qualquer $a, b, c \in A$, as seguintes regras de inferências valem*

$$\frac{}{a \to_{R*} a}\ \textit{(refl)}$$

$$\frac{a \to_R b \quad b \to_{R*} c}{a \to_{R*} c}\ \textit{(rtrans)}$$

```
Inductive refltrans {A:Type} (R: Rel A) : A → A → Prop :=
| refl: ∀ a, (refltrans R) a a
| rtrans: ∀ a b c, R a b → refltrans R b c → refltrans R a c.
```

Tabela 3.1: Definição das regras de inferência de fecho transitivo-reflexivo em Coq.

Lema 3.1. *Dada uma relação R, se* $a \rightarrow_{R*} b$ *e* $b \rightarrow_{R*} c$*, então* $a \rightarrow_{R*} c$*.*

A prova deste lema é feita por indução em $a \rightarrow_{R*} b$*. Isto gera dois casos, o caso da reflexividade pode ser trivialmente fechado. O caso geral depende da aplicação da definição 3.1 seguida da aplicação da hipótese de indução.*

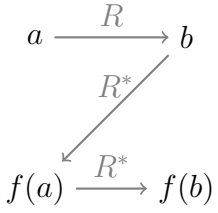
```
Lemma refltrans_composition {A} (R: Rel A): ∀ t u v, refltrans R t u →
refltrans R u v → refltrans R t v.
Proof.
  intros t u v.
  intros H1 H2.
  induction H1.
  - assumption.
  - apply rtrans with b.
    + assumption.
    + apply IHrefltrans; assumption.
Qed.
```

Tabela 3.2: Prova do Lema 3.1 em Coq.

A definição do que é um SRA e o que é o fecho transitivo-reflexivo permite definir o que é a propriedade Z. A propriedade Z diz respeito a um conjunto de regras que o SRA deve ter para se encaixar nesta propriedade. O valor de demonstrar a propriedade Z sobre um sistema está em que, essencialmente, se um SRA qualquer atende a propriedade Z então este é confluente [25]. A demonstração da confluência do cálculo λ_x pode ser feita demonstrando que este, um SRA, atende a propriedade Z.

Definição 3.2. *Um SRA* (A, R)*, qualquer, é Z se existe um mapeamento* $f : A \rightarrow A$ *tal que* $\forall a$ *e b,* $a \rightarrow_R b$*, então* $b \rightarrow_{R*} f(a) \ \wedge \ f(a) \rightarrow_{R*} f(b)$

Intuitivamente, o seguinte diagrama vale (de onde origina-se o nome):



```
Definition Z_prop {A:Type} (R: Rel A) := ∃ f:A → A, ∀ a b, R a b →
((reftrans R) b (f a) ∧ (reftrans R) (f a) (f b)).
```

Tabela 3.3: Definição da Propriedade Z para relações em Coq.

Além disso, a função f é Z se esta é função de mapeamento de algum SRA que atende Z:

```
Definition f_is_Z {A:Type} (R: Rel A) (f: A → A) := ∀ a b, R a b →
((reftrans R) b (f a) ∧ (reftrans R) (f a) (f b)).
```

Tabela 3.4: Definição da Propriedade Z para funções de mapeamento em Coq.

Definição 3.3. *Dadas relações binárias R e R' sobre um conjunto A qualquer, $f : A \to A$, função, é Z fraca para R por R' se $\forall a, b \in A$, $a \to_R b$ implica $b \to_{R'*} f(a) \ \wedge \ f(a) \to_{R'*} f(b)$*

```
Definition f_is_weak_Z {A} (R R': Rel A) (f: A → A) := ∀ a b, R a b →
((reftrans R') b (f a) ∧ (reftrans R') (f a) (f b)).
```

Tabela 3.5: Definição da Propriedade Z fraca para funções de mapeamento em Coq.

A propriedade Z pode ser expressa na forma Z fraca para uma relação R por ela mesma. Essa definição pode ser usada para compor outras definições.

Definição 3.4. *Dado um SRA (A, R) onde $R = R_1 \cup R_2$. Se existe mapeamentos $f_1, f_2 : A \to A$ tais que:*

1. *f_1 é Z para R_1.*
2. *$a \to_{R_1} b$ implica $f_2(a) \to_{R*} f_2(b)$.*
3. *$a \to_{R*} f_2(a)$ vale para qualquer $a \in Im(f_1)$.*
4. *$f_2 \circ f_1$ é Z fraca para R_2 por R.*

Então (A, R) é Z Composicional.

```
Definition comp {A} (f1 f2: A → A) := fun x:A ⇒ f1 (f2 x).
Notation "f1 # f2":= (comp f1 f2).
Definition Z_comp {A:Type} (R :Rel A) := ∃ (R1 R2: Rel A) (f1 f2: A →
A), (∀ x y, R x y ↔ (R1 !_! R2) x y) ∧ f_is_Z R1 f1 ∧ (∀ a b, R1 a b →
(refltrans R) ((f2 # f1) a) ((f2 # f1) b)) ∧ (∀ a b, b = f1 a → (refltrans R)
b (f2 b)) ∧ (f_is_weak_Z R2 R (f2 # f1)).
```

Tabela 3.6: Definição da Propriedade Z Composicional em Coq.

Lema 3.2. *Dado um SRA (A, R), se (A, R) é Z Composicional, então (A, R) é Z.*

É necessário provar que existe um mapeamento f, tal que f é Z. Tendo f_1 e f_2 funções de mapeamento de (A, R) para com a Z composicional, se provado que $f_2 \circ f_1$ é o mapeamento tal qual $f_2 \circ f_1$ é Z, então (A, R) é Z. Dado que $R = R_1 \cup R_2$ há dois casos a considerar:

1. *$a \rightarrow_{R_1} b$: Utilizando o lema 3.1, na prova que $b \rightarrow_{R*} f_2 \circ f_1(a)$ é necessário demonstrar que $b \rightarrow_{R_1*} f_1(a)$ e $f_1(a) \rightarrow_{R*} f_2 \circ f_1(a)$. A prova do primeiro caso é verificada pelo fato de f_1 ser Z para R_1. Já o segundo caso da conjunção é explicado por $f_1(a) \in Im(f_1)$, logo, por definição da Z composicional, esta redução ocorre.*
2. *$a \rightarrow_{R_2} b$: Esse caso é trivial uma vez que pela hipótese que R é Z Composicional, $f_2 \circ f_1$ é Z fraca para R_2 por R.*

```
Theorem Z_comp_implies_Z_prop {A:Type}: ∀ (R :Rel A), Z_comp R →
Z_prop R.
Proof.
  intros R H.
  unfold Z_prop. unfold Z_comp in H. destruct H as
  [ R1 [ R2 [f1 [f2 [Hunion [H1 [H2 [H3 H4]]]]]]]].
  ∃ (f2 # f1).
  intros a b HR.
  apply Hunion in HR. inversion HR; subst. clear HR.
  - split.
    + apply refltrans_composition with (f1 a).
      × apply H1 in H.
        destruct H as [Hb Hf].
        apply (refltrans_union R1 R2) in Hb.
        apply refltrans_union_equiv with R1 R2.
        ** apply Hunion.
        ** apply Hb.
      × apply H3 with a; reflexivity.
    + apply H2; assumption.
  - apply H4; assumption.
Qed.
```

Tabela 3.7: Prova do Lema 3.2 em Coq.

Definição 3.5. *Dado um SRA* (A, R) *onde* $R = R_1 \cup R_2$*. Se existe mapeamentos* $f_1, f_2 : A \to A$ *tais que:*

1. $a \to_{R_1} b$ *implica* $f_1(a) = f_1(b)$
2. $a \to_{R_1*} f_1(a)$*, para todo* a*.*
3. $a \to_{R*} f_2(a)$ *vale para qualquer* $a \in Im(f_1)$*.*
4. $f_2 \circ f_1$ *tem a propriedade Z fraca para* R_2 *e* R*.*

Então (A, R) *é Z Composicional com Igualdade.*

```
Definition Z_comp_eq {A:Type} (R :Rel A) := ∃ (R1 R2: Rel A) (f1 f2: A
→ A), (∀ x y, R x y ↔ (R1 !_! R2) x y) ∧ (∀ a b, R1 a b → (f1 a) = (f1
b)) ∧ (∀ a, (refltrans R1) a (f1 a)) ∧ (∀ b a, a = f1 b → (refltrans R) a (f2
a)) ∧ (f_is_weak_Z R2 R (f2 # f1)).
```

Tabela 3.8: Definição de Z Composicional com Igualdade em Coq.

Lema 3.3. *Dado um SRA* (A, R)*, Se* R *é Z Composicional com Igualdade, então* R *é Z Composicional.*

Dado que (A, R) *é Z Composicional com Igualdade, é necessário provar a partir das propriedades de Z Composicional com Igualdade as premissas de Z Composicional. Tendo* f_1 *e* f_2 *como funções de mapeamento de* (A, R) *para com a Z Composicional com Igualdade. Se, valendo-se das premissas de Z módulo, provar as premissas de Z composicional, então* R *é Z composicional. São quatro ramos de prova a considerar:*

1. f_1 *é Z para* R_1*: Pela definição de* f_1 *ser Z,* $a \to_{R_1} b$*. Por consequência, pela premissa 1 de Z módulo,* $f_1(a) = f_1(b)$*. Logo, para provar que* $b \to_{R_1*} f_1(a) \wedge f_1(a) \to_{R_1*} f_1(b)$*, basta substituir as ocorrências de* $f_1(a)$*. O lado esquerdo da conjunção é a própria premissa 2 da definição 3.5 e o lado direito é trivialmente provado pela reflexividade definida em 3.1.*
2. $a \to_{R_1} b$ *implica* $f_2(f_1(a)) \to_{R*} f_2(f_1(b))$*: Novamente, valendo-se do fato que* $f_1(a) = f_1(b)$ *a prova é trivial por substituição de* $f_1(a)$ *e aplicação da reflexividade.*
3. $a \to_{R*} f_2(a)$ *vale para qualquer* $a \in Im(f_1)$*: Esta condição faz parte da própria definição de Z módulo. Logo, é trivialmente provado.*
4. $f_2 \circ f_1$ *é Z fraca para* R_2 *por* R*: Similar ao caso anterior, este caso pode ser concluído pela definição 3.5.*

```
Lemma Z_comp_eq_implies_Z_comp {A:Type}: ∀ (R : Rel A), Z_comp_eq R
→ Z_comp R.
Proof.
  intros R Heq. unfold Z_comp_eq in Heq.
  destruct Heq as [R1 [R2 [f1 [f2 [Hunion [H1 [H2 [H3 H4]]]]]]]].
  unfold Z_comp.
  ∃ R1, R2, f1, f2.
  split.
  - assumption.
  - split.
    + unfold f_is_Z.
      intros a b H; split.
      × apply H1 in H. rewrite H. apply H2.
      × apply H1 in H. rewrite H. apply refl.
    + split.
      × intros a b H.
        unfold comp.
        apply H1 in H.
        rewrite H.
        apply refl.
      × split; assumption.
Qed.
```

Tabela 3.9: Prova do Lema 3.3 em Coq.

Corolário 3.1. *Dado um SRA (A, R), Se R é Z Composicional com Igualdade, então R é Z.*

Se (A, R) é Z Composicional com Igualdade, então é Z Composicional pelo lema 3.3. Logo, (A, R) é Z pelo lema 3.2.

```
Corollary Z_comp_eq_implies_Z_prop {A:Type}: ∀ (R : Rel A), Z_comp_eq
R → Z_prop R.
Proof.
  intros.
  apply Z_comp_eq_implies_Z_comp in H.
  apply Z_comp_implies_Z_prop.
  assumption.
Qed.
```

Tabela 3.10: Prova do Corolário 3.1 em Coq.

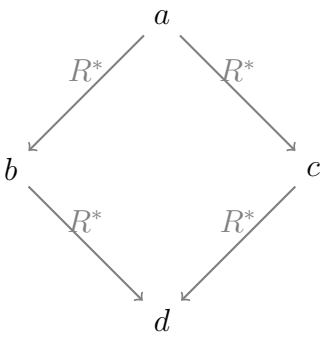
3.2 Confluência

A confluência é uma propriedade sobre sistema de reduções abstratos tal qual: Dado um SRA (A, R) e um termo t, a escolha das reduções possíveis em R a serem aplicada sobre t não modifica o resultado final. Dentre as escolhas de regras possíveis, as divergências sempre resultam em um termo comum após a aplicação de um conjunto - possivelmente vazio - de regras. Um exemplo de sistema de reescrita confluente é a aritmética básica, respeitando os delimitadores e as regras de precedência dos operadores, o resultado

da expressão sempre será o mesmo não importando a ordem a qual as sub-expressões são resolvidas. A confluência é uma propriedade importante sobre SRAs pois, permite heurísticas distintas de reduções. Mais formalmente, confluência pode ser definida como:

Definição 3.6. *Dado um SRA* (A, R)*,* (A, R) *é confluente quando* $\forall a, b$ *e* c*,* $a \rightarrow_{R*} b \wedge a \rightarrow_{R*} c$ *implica* $\exists d, b \rightarrow_{R*} d \wedge c \rightarrow_{R*} d$*.*

Intuitivamente, (A, R) *é confluente quando o diagrama abaixo vale:*



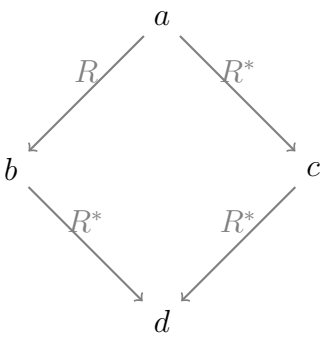

```
Definition Confl {A:Type} (R: Rel A) := ∀ a b c, (refltrans R) a b →
(refltrans R) a c → (∃ d, (refltrans R) b d ∧ (refltrans R) c d).
```

Tabela 3.11: Definição de Confluência em Coq.

Como exposto anteriormente, a propriedade Z é importante pela sua relação com a confluência. Um SRA que atende esta propriedade é confluente [26]. A demonstração dessa proposição é densa em relação ao que já foi apresentado neste trabalho. Em razão disso, define-se a noção de semi-confluência, conceito equivalente a noção de confluência, e demonstra-se que a propriedade Z implica a semi-confluência.

Definição 3.7. *Dado um SRA* (A, R)*,* (A, R) *é semi-confluente quando* $\forall a, b$ *e* c*,* $a \rightarrow_{R} b \wedge a \rightarrow_{R*} c$ *implica* $\exists d, b \rightarrow_{R*} d \wedge c \rightarrow_{R*} d$*.*

Intuitivamente, (A, R) *é semi-confluente quando o diagrama abaixo vale:*



Lema 3.4. *Dado um SRA S, se S é semi-confluente, então S é confluente.*

A construção dessa prova se dá por indução na premissa $a \rightarrow_R b$ da definição de semi-confluência. A partir daí todos os casos são triviais.

```
Lemma SemiConfl_imples_Confl{A: Type}: ∀ R:Rel A, SemiConfl R → Confl
R.
Proof.
  unfold Confl.
  unfold SemiConfl.
  intros.
  generalize dependent c.
  induction H0.
  - intros.
    ∃ c.
    split.
    × assumption.
    × apply refl.
  - intros.
    specialize (H a b c0).
    apply H in H0.
    + destruct H0.
      destruct H0.
      apply IHrefltrans in H0.
      destruct H0.
      destruct H0.
      ∃ x0.
      split.
      × assumption.
      × apply refltrans_composition with x; assumption.
    + assumption.
Qed.
```

Tabela 3.12: Prova do Teorema 3.4 em Coq.

Lema 3.5. *Dado um SRA S, se S é confluente, então S é semi-confluente.*

A prova deste lema se dá diretamente pela definição de confluência. Se S é confluente, para qualquer a, b e c, tal que $a \rightarrow b$ e $a \rightarrow_* c$, tomando o ponto de convergência $d = a$, a resolução é trivial.

```
Lemma Confl_imples_SemiConfl{A: Type}: ∀ R:Rel A, Confl R→ SemiConfl
R.
Proof.
  unfold Confl.
  unfold SemiConfl.
  intros.
  apply H with a.
  - apply rtrans with b.
    + assumption.
    + apply refl.
  - assumption.
Qed.
```

Tabela 3.13: Prova do Teorema 3.5 em Coq.

Teorema 3.1. *Dado um SRA S, S é semi-confluente se, e somente se, S é confluente.*

Este teorema é consequência direta dos lemas 3.4 e 3.5.

```
Theorem Semi_equiv_Confl {A: Type}: ∀ R: Rel A, Confl R ↔ SemiConfl R.
Proof.
  intros.
  split.
  - apply Confl_imples_SemiConfl.
  - apply SemiConfl_imples_Confl.
Qed.
```

Tabela 3.14: Prova do Teorema 3.1 em Coq.

Teorema 3.2. *Dado um SRA S, se S é Z, então S é semi-confluente.*

Supondo a, b e c e função mapeamento f tal que $a \to b$. Pela propriedade Z, $a \to_* f(a)$. Induzindo na hipótese $a \to_* c$, dois casos são gerados. O caso base, onde $a \to c'$, pela propriedade Z, $c \to_* f(a)$, logo, é verificada a semi-confluência para o caso base. E o caso geral, onde $a \to_* c$, neste caso em um ou mais passos. A divergência a ser analisada é $c' \to_* c \wedge c' \to_* f(a)$. Isto pode ser concluído por hipótese de indução.

```
Theorem Z_prop_implies_SemiConfl {A:Type}: ∀ R: Rel A, Z_prop R → SemiConfl R.
Proof.
  intros R HZ_prop.
  unfold Z_prop in HZ_prop.
  unfold SemiConfl.
  destruct HZ_prop.
  intros a b c Hrefl Hrefl'.
  assert (Haxa: refltrans R a (x a)).
  { apply rtrans with b. - assumption. - apply H. assumption. }
  apply H in Hrefl.
  destruct Hrefl.
  clear H1.
  generalize dependent b.
  induction Hrefl'.
  - intros.
    ∃ (x a).
    split; assumption.
  - intros.
    destruct IHHrefl' with b0.
    + apply refltrans_composition with (x a); apply H; assumption.
    + apply refltrans_composition with (x b).
      × apply refltrans_composition with (x a).
        ** assumption.
        ** apply H.
           assumption.
      × apply refl.
    + ∃ x0.
      assumption.
Qed.
```

Tabela 3.15: Prova do Teorema 3.2 em Coq.

Corolário 3.2. *Dado um SRA S, se S é Z, então S é confluente.*

Este corolário é consequência direta dos teoremas 3.1 e 3.2

```
Corollary Zprop_implies_Confl_via_SemiConfl {A:Type}: ∀ R: Rel A,
Z_prop R → Confl R.
Proof.
  intros.
  apply Semi_equiv_Confl.
  apply Z_prop_implies_SemiConfl.
  assumption.
Qed.
```

Tabela 3.16: Prova do Corolário 3.2 em Coq.

Corolário 3.3. *Dado um SRA S, se S é Z Composicional, então S é confluente.*

Este corolário é consequência direta do lema 3.2 e do teorema 3.2

```
Corollary Z_comp_is_Confl {A}: ∀ (R: Rel A), Z_comp R → Confl R.
Proof.
  intros R H.
  apply Z_comp_implies_Z_prop in H.
  apply Zprop_implies_Confl_via_SemiConfl.
  assumption.
Qed.
```

Tabela 3.17: Prova do Corolário 3.3 em Coq.

Corolário 3.4. *Dado um SRA S, se S é Z Composicional com Igualdade, então S é confluente.*

Este corolário é consequência direta do corolário 3.1 e do teorema 3.2

```
Corollary Z_comp_eq_is_Confl {A: Type}: ∀ (R: Rel A), Z_comp_eq R →
Confl R.
Proof.
    intros.
    apply Z_comp_eq_implies_Z_prop in H.
    apply Zprop_implies_Confl_via_SemiConfl.
    assumption.
Qed.
```

Tabela 3.18: Prova do Corolário 3.4 em Coq.

3.3 Estratégia da Prova de Confluência do Cálculo λ_x

O resultado do Corolário 3.4 é interessante para a prova da confluência do cálculo λ_x. Neste caso, para construir a prova da confluência deste sistema de reduções, basta encontrar mapeamentos f_1 e f_2 que satisfaçam as premissas da Z Composicional com Igualdade. Em [7], K. Nakazawa define os mapeamentos P e B para este fim.

Definição 3.8. *Mapeamentos M^P e M^B*

$$\begin{aligned} x^P &= x \\ (\lambda x.M)^P &= \lambda x.M^P \\ (MN)^P &= M^P N^P \\ (M\langle x := N\rangle)^P &= M^P[x := N^P] \end{aligned} \qquad \begin{aligned} x^B &= x \\ (\lambda x.M)^B &= \lambda x.M^B \\ ((\lambda x.M)N)^B &= M^B[x := N^B] \\ (MN)^B &= M^B N^B \\ (M\langle x := N\rangle)^B &= M^B\langle x := N^B\rangle \end{aligned}$$

A premissa do mapeamento P é trocar as substituições explícitas por substituições implícitas. Estas são as meta-substituições do cálculo original, apresentadas na seção 2.2.2. A meta-substituição para o mapeamento P é assumida, isto é, os detalhes de como esta ocorre não é importante para a argumentação. Isto tem algumas implicações que serão discutidas na próxima seção.

O mapeamento B segue a premissa de disparar a simulação da beta-redução. Quando composta com P, na forma $B \circ P$, o mapeamento resultante torna-se a função de mapeamento para a propriedade Z. Nesta configuração, a regra para substituições explícitas de B jamais é aplicada, mantida apenas pelo rigor do formalismo. A composição dos mapeamentos P e B é denotada como M^{PB}.

Tomando a definição 2.7 de substituições em λ_x, o autor decompõe esta relação em duas outras. As relações β_x e π. A primeira relação apenas possui a regra que dispara a simulação da beta. A segunda possui as demais regras, responsáveis por completa-la. Essa decomposição é interessante para compor a prova.

Definição 3.9. *Regras de β_x e de π, decomposição das regras de λ_x, onde $x \neq y$ e $x \notin FV(N)$*

- $(\lambda x.M)N \to_{\beta_x} M\langle x := N\rangle$
- $y\langle y := N\rangle \to_{\pi(hit)} N$

- $x\langle y := N\rangle \rightarrow_{\pi(gc)} x$
- $(\lambda x.P)\langle y := N\rangle \rightarrow_{\pi(abs)} \lambda x.P\langle y := N\rangle$
- $(PQ)\langle y := N\rangle \rightarrow_{\pi(app)} P\langle y := N\rangle Q\langle y := N\rangle$

Sendo assim, provar a confluência do cálculo λ_x reduz-se a provar as quatro premissas da Z Composicional com Igualdade. As premissas com os mapeamentos P e B, e com as relações β_x e π podem ser escritas da seguinte forma:

Definição 3.10. *Lemas para demonstrar λ_x como Z Composicional com Igualdade*

1. $M \rightarrow_\pi N$ *implica* $M^P = N^P$.
2. $M \rightarrow_{\pi *} M^P$, *para todo* M.
3. $M \rightarrow_{\lambda_x} M^B$, $M \in Im(P)$ - *isto é: puro, sem substituições explícitas.*
4. $M \rightarrow_{\beta_x} N$ *implica* $N \rightarrow_{\lambda_x *} M^{PB} \rightarrow_{\lambda_x *} N^{PB}$.

A partir desses lemas, o autor realiza a argumentação que conclui a prova. Essencialmente por indução na estrutura de M ou indução na hipótese do enunciado. Transpondo para este trabalho, é necessário definir uma forma de representar o cálculo dentro do ambiente Coq. Uma possibilidade para equacionar este problema é o *framework Locally Nameless Representation*.

3.3.1 O *framework Locally Nameless Representation*

O assistente de provas Coq utiliza da Lógica de Predicados Construtivista nas demonstrações. Todas as propriedades tem que ser construídas a partir das hipóteses. Isso implica que todas as noções até então apresentadas sobre o cálculo devem concretamente existir. Para suprir essa necessidade, é possível utilizar o *framework* introduzido por A. Charguéraud. Este *framework* define diversas das noções à respeito do Cálculo Lambda com implementação prática [27].

Esse framework é nomeado *Locally Nameless Representation*, do inglês Representação Localmente Sem Nomes. A ideia do *framework* é implementar o Cálculo Lambda com Índices de De Bruijn para as variáveis ligadas e com nomes para as variáveis livres. Esse esquema propõe não precisar lidar com a alfa-conversão de termos e ao mesmo tempo não ter que lidar com operações de *shifting* dos índices. Isso torna a implementação de substituições consideravelmente mais simples.

Nesta representação, dois termos podem ser por definição alfa-equivalentes mas sintaticamente diferentes. Isto ocorre porquê esta representação não é estritamente equivalente

ao Cálculo Lambda original. A gramática do original do cálculo é expandida com um novo construtor que separa as variáveis ligadas das variáveis livres. Por questão de ordem prática, a *Locally Nameless Representation* já será apresentada considerando também a expansão proposta pelo Cálculo Lambda com Substituições Explícitas.

Definição 3.11. *Gramática Locally Nameless Representation expandida com Substituições Explícitas.*

$$M := x \mid i \mid \lambda M \mid MM \mid M\langle M \rangle$$

Uma característica importante deste *framework* para este trabalho é que obter o conjunto de variáveis livres de um termo t qualquer é facilitado. Apenas as variáveis livres do termo são nominais, a tarefa de obtê-las reduz-se a buscar nomes em t. A implementação tanto desta função quanto de uma função $fresh$ - responsável por retornar uma variável nominal que não ocorre no corpo de t. O conjunto de variáveis livres pode ser obtida via a seguinte função:

Definição 3.12. *Conjunto de variáveis livres em Locally Nameless Representation.*

$$FV_c(t) = \begin{cases} \emptyset & , \ t \ \textit{índice} \\ \{t\} & , \ t \ \textit{variável livre} \\ FV_c(t_1) \cup FV_c(t_2) & , \ t = t_1 \ t_2 \\ FV_c(t_1) & , \ t = \lambda \ t_1 \\ FV_c(t_1) \cup FV_c(t_2) & , \ t = t_1\langle t_2 \rangle \end{cases}$$

Este *framework* não considera termos com índices soltos. Um índice sempre deve estar atrelado a uma abstração. Contudo, dada uma abstração λt_1 é possível que o corpo t_1, isoladamente, tenha ocorrências livres de índices. Quando é necessário inspecionar o corpo dessa abstração, as variáveis desta abstração são substituídas por uma variável nominal não pertencente ao conjunto de variáveis livres de t_1.

Esse processo de substituir a variável ligada por uma variável livre é denominado como abertura do termo. Para isto ocorrer, é mantido um índice k sempre inicializado com zero. Esse índice mantém a contagem de quantos níveis de abstração estão sendo inspecionadas no processo de substituições. Isto permite a substituição da variável ligada no processo de beta-redução. Esta função de abertura pode ser definida recursivamente como:

Definição 3.13. *Abertura de termos*

$$\{k \rightsquigarrow x\}\ t = \begin{cases} x & , t \text{ índice e } k = t \\ t & , t \text{ índice e } k \neq t \\ t & , t \text{ variável livre} \\ \lambda\ \{k+1 \rightsquigarrow x\}\ t_1 & , t = \lambda\ t_1 \\ \{k \rightsquigarrow x\}\ t_1\ \{k \rightsquigarrow x\}\ t_2 & , t = t_1\ t_2 \\ \{k+1 \rightsquigarrow x\}\ t_1\langle\{k \rightsquigarrow x\}\ t_2\rangle & , t = t_1\langle t_2\rangle \end{cases}$$

A prova de K. Nakazawa assume a meta-substituição para concluir a prova do teorema proposto. No contexto de uma prova construtivista, este passo não é possível . A meta-substituição precisa ser concretamente definida. É necessário então definir o que é a meta-substituição para seguir com a abordagem do autor.

Felizmente, a própria definição 3.13 implementa esta noção de meta-substituição para esta versão do cálculo. Esta implementa as regras de reescrita dos termos para substituições geradas por beta-reduções. O trabalho pode valer-se dessa implementação e do *framework* como um todo para a construção das provas. Em tese, a prova seguindo a estratégia de K. Nakazawa pode ser realizada.

Capítulo 4

Discussão de Resultados

Neste capitulo, serão apresentados os resultados obtidos no trabalho. Será discutida a modelagem da prova da confluência do cálculo λ_x junto à *Locally Nameless Representation* em Coq. Serão expostos os principais desafios para com a prova, em especial até que ponto avançou-se na formalização e o porquê esta não pôde ser concluída. Em ligação a este último tópico, serão discutidas também outras possibilidades para a conclusão da prova.

A modelagem da prova passa pela definição completa do que é o λ_x no assistente de provas Coq. Diferente da relação da confluência com a propriedade Z, em que as construções são lineares e relativamente simples de expressar em texto, a modelagem do λ_x é extensa e bastante densa. Explica-la em detalhes neste documento resultaria em capítulos e mais capítulos apenas apresentando definições. Além disso, tais definições apresentam certa volatilidade, por mais que haja uma definição formal do sistema, este pode ser escrito de diversas formas, cada qual com suas conveniências para com a estratégia de prova. Daqui em diante, apenas os lemas centrais serão expostos. O leitor que desejar investigar este trabalho pode acessar o repositório em https://github.com/nunesgrf/lx-confluence.

Diferente dos capítulos anteriores, onde uma abordagem *bottom-up* de apresentação dos lemas ocorreu. Esta seção abordará as provas de forma *top-down*. O Teorema da Confluência para o cálculo λ_x será apresentada primeiro e as condicionantes discutidas. A razão para isto é trazer visibilidade para os pontos onde a formalização não pôde avançar e discutir os desafios relacionados aos mesmos.

4.1 A Construção da Prova em Coq

A construção da prova via Coq inicia-se no objetivo maior deste trabalho. Formalizar a confluência do cálculo λ_x pelos passos de K. Nakazawa. A relação da Confluência com a propriedade Z é bem definida pelo Corolário 3.2. Desta forma, o problema reduz-se a provar que λ_x é Z Composicional com Igualdade.

.

Teorema 4.1. λ_x *é confluente.*

```
Theorem lambda_x_is_confluent: Confl lx.
Proof.
  apply Z_prop_implies_Confl.
  apply Z_comp_eq_implies_Z_prop.
  apply lambda_x_Z_comp_eq.
Qed.
```

Tabela 4.1: Prova do Teorema 4.1 em Coq.

A prova da Z Composicional com Igualdade para o λ_x reduz às premissas desta Z (3.10). As quatro definições são o suficiente para esta inferência. Há de se buscar lemas auxiliares para que a construção torne-se direta. Desta forma, a correção do lema é encontrada mediante a correção destes lemas.

Lema 4.1. λ_x *é* Z *Composicional com Igualdade.*

```
Theorem lambda_x_Z_comp_eq: Z_comp_eq lx.
Proof.
  unfold Z_comp_eq.
  ∃ x_ctx, b_ctx, P, B. split.
  - intros x y; split.
    + intro HBx.
      apply union_or.
      inversion HBx; subst.
      × right; assumption.
      × left; assumption.
    + intro Hunion.
      apply union_or in Hunion.
      inversion Hunion.
      × apply x_ctx_rule. assumption.
      × apply b_ctx_rule. assumption.
  - split.
    + apply lambda_x_redution_implies_P_eq.
    + split.
      × apply t_reduces_to_P_t.
      × split.
        ** intros b a Heq.
           apply pure_term_implies_red_to_B.
           rewrite → Heq.
           apply img_of_P_is_pure.
        ** apply B_comp_P_is_weak_Z_for_B_by_lx.
Qed.
```

Tabela 4.2: Prova do Teorema 4.1 em Coq.

Os lemas auxiliares indicados por K. Nakazawa dizem respeito a uma série de propriedades envolvendo a pureza dos termos i.e. a ausência da substituições explícitas. Combinados entre si, esses lemas são usado para discorrer sobre a validade das proposições

do λ_x. Os lemas, de forma geral, são simples e intuitivos mas, representam construções densas via assistente de provas Coq. Os lemas definem-se como:

1. M^P é termo puro.
2. Se M é puro, $M^P = M$.
3. Se M é puro, $M\langle x := N\rangle \to_{\pi *} M[x := N]$.
4. $M \to_\pi N$ implica $M^P = N^P$.

Inicialmente, para este trabalho, a noção de termo puro era de termos do *framework locally nameless* sem substituições explícitas. Esta abordagem compunha a abordagem da ferramenta sobre termos e a premissa de P. A análise sobre essa composição para os lemas expõe algumas complexidades. Em razão disso a definição foi flexibilizada para abordar somente a ausência de substituições explícitas.

A decisão por esta alteração partiu da necessidade de resolver sequentes envolvendo meta-substituições, substituições explícitas e a função P para termos gerais. Basicamente, era necessário demonstrar que dado um termo t com meta-substituição, a propagação de P para dentro da meta-substituição não incorreria em prejuízos para o resultado final. O que por si só é razoável. A complexidade cresce quando t é um termo com substituição explícita. Ao propagar a P sobre este termo e sobre a meta-substituição, o resultado é uma composição de meta-substituições. Ao tornar a definição mais flexível, esse caso não ocorre mais.

Esta alteração permitiu construções mais simples. Cessou-se a necessidade de lidar com múltiplas meta-substituições e com a abertura do termo quando este é puro. Em contrapartida, esta redefinição distanciou-se das noções de *locally nameless* derrubando uma série de lemas que poderiam vir a ser importantes na continuidade da prova. Mas também permitiu a construção do seguinte resultado:

Lema 4.2. *Se t_1 e t_2 são termos puros, para qualquer n, $\{n \rightsquigarrow t_2\}\ t_1$ é termo puro.*

A prova deste lema ocorre por indução na premissa que t_1 é termo. Os ajustes na construção da prova passam por generalizar o valor de n na hipótese de indução, dada a natureza incremental deste valor na definição 3.13. E também, a análise sobre a contradição das hipóteses no caso da substituição explícita.

```
Lemma lterm_msub : ∀ t1 t2 n, lterm t1 → lterm t2 → lterm ({n ⇝ t2}t1).
Proof.
  intros.
  generalize dependent n.
  induction t1.
  - intros.
    simpl.
    destruct (n0 =? n).
    + destruct (n =? n0).
      × assumption.
      × assumption.
    + destruct (n =? n0).
      × assumption.
      × assumption.
  - intros.
    simpl.
    apply lterm_var.
  - intros.
    simpl.
    apply lterm_app.
    + apply IHt1_1.
      inversion H.
      assumption.
    + apply IHt1_2.
      inversion H.
      assumption.
  - intros.
    simpl.
    apply lterm_abs.
    apply IHt1.
    inversion H.
    assumption.
  - inversion H.
Qed.
```

Tabela 4.3: Prova do Lema 4.2 em Coq.

Este resultado é importante para a construção da noção de pureza para M, $M \in Im(P)$. No caso da substituição explícita, a P aplica a transformação para a meta e o Lema 4.2 viabiliza a aplicação das hipóteses de indução. A construção desta noção de pureza é trivial para todos os demais casos. Pode ser expressa pelo seguinte lema.

Lema 4.3. *para todo M, M^P é um termo puro.*

Pela definição de P e de termos puros, a construção desse lema por indução é simples. Este lema pode ser utilizado posteriormente para concluir o resultado que $M \rightarrow_{\lambda_{x}*} M^B$, $M \in Im(P)$. Uma das premissas para λ_x ser Z Composicional com igualdade.

```
Lemma img_of_P_is_pure: ∀ t, lterm (P t).
Proof.
  intros t.
  induction t.
  - simpl. apply lterm_bvar.
  - simpl. apply lterm_var.
  - simpl. apply lterm_app.
    + apply IHt1.
    + apply IHt2.
  - simpl. apply lterm_abs.
    apply IHt.
  - simpl.
    apply lterm_sub.
    + apply IHt1.
    + apply IHt2.
Qed.
```

Tabela 4.4: Prova do Lema 4.3 em Coq.

Essa noção de pureza também permite concluir com facilidade o predicado de que P não tem efeitos sobre termos puros. Assim como o lema anterior, a análise reduz ser a induzir sobre a estrutura do termo. Como o caso da substituição explícita não precisa ser analisado, a conclusão é direta. A formalização deste resultado pode ser vista no lema a seguir.

Lema 4.4. *Se M é puro, $M^P = M$.*

```
Lemma lterm_t_implies_P_t_eq_t: ∀ t, lterm t → P t = t.
Proof.
  induction 1.
  - simpl.
    reflexivity.
  - simpl.
    reflexivity.
  - simpl.
    rewrite IHlterm1.
    rewrite IHlterm2.
    reflexivity.
  - simpl.
    rewrite IHlterm.
    reflexivity.
Qed.
```

Tabela 4.5: Prova do Lema 4.4 em Coq.

O predicado do item 3 não foi investigado por este trabalho. O período limitado para a compreensão da teoria como um tudo, a definição de um escopo e a construção em Coq das bases para a prova do Lema não permitiu a abordagem desta proposição por este trabalho. Algum desenvolvimento foi preliminar foi realizado porém, não digno de nota. Esse resultado é usado para com a demonstração que a composição de B e P é Z fraca.

Por consequência, a asserção que o mapeamento M^{PB} é Z fraca para β_x por λ_x também não avançou. A construção desse lema é complexa e envolve a boa definição das regras de λ_x como sistema de reduções. A definição desse conjunto de regras foi volátil ao longo do desenvolvimento deste trabalho, a implementação não é uma tradução direta do modelo teórico. Há de se primeiro definir o conjunto de regras conveniente para os demais lemas e então abordar este problema, dada a sua natureza complexa.

Por fim, o item 4 dos lemas propostos por K. Nakazawa. O lema enuncia que se existe uma redução em um passo de π entre M e N, então $M^P = N^P$. Este lema é a razão para utilizar-se da definição de Z Composicional com Igualdade, este já escreve diretamente uma das premissas para λ_x atender esta propriedade. Não há restrições sobre os termos M e N. Portanto, a análise dos casos também passa pela análise de substituições explícitas.

Este lema discorre sobre como as regras de λ_x reduzem os termos e a relação disto com P. A análise deste lema é feita por indução na hipótese. E então, é necessário demonstrar para cada caso que a aplicação de P no elemento lado esquerdo da redução é igual a aplicar no elemento do lado direito, versão já reduzida. Com as presentes definições de regras de substituição explícita e meta-substituição, isto não é possível. As definições não expressam os mesmos passos.

Um passo de contorno para esta questão é a reavaliação de ao menos uma das definições. Por um lado, se redefinida a noção de substituição explícita expressa por K. Nakazawa, este trabalho deixa de verificar a prova proposta pelo autor. Pelo outro lado, redefinir a noção de meta-substituição de A. Charguéraud significa perda do *framework* como um todo para este trabalho. A operação de abertura de termos é central para os lemas que dali seguem. Conjectura-se então que a *locally nameless representation* não pode ser utilizada para formalizações em λ_x.

Agora supondo que este problema possa ser contornado. Em certos lemas auxiliares para este item, emerge a necessidade de compor meta-substituições. A propagação de P para dentro de meta-substituições é necessária para concluir alguns ramos da prova. A questão aparece quando os termos possuem substituições explícitas. A propagação de P implica no surgimento de tais composições. Definiu-se enunciados para casos particulares da composição, suficiente para concluir a prova.

Utilizando desses enunciados, as provas poderiam seguir. Mas para completa-las, estes enunciados também precisariam serem completados. A estratégia utilizada foi em investir primeiro nestes ramos em aberto para evitar retrabalho. Essa investigação levou a uma conclusão a respeito da noção de meta-substituição proposta pelo *framework*. Não é possível, em um caso geral, compor as meta-substituições. Isto ficou denominado como O Problema da Composicionalidade nas iterações ao longo do desenvolvimento.

4.2 O Problema da Composicionalidade

A meta-substituição do Cálculo-Lambda assume a composicionalidade. Isto é, dadas duas meta-substituições sobre um mesmo termo t, é possível realizar uma composição das mesmas de forma a inverter a ordem a qual as variáveis livres são substituídas sem prejuízo para o resultado final. O Lema da Substituição enuncia essa propriedade. As condições para essa composição são apenas duas, as variáveis substituídas pelas meta-substituições tem que ser diferentes e também a variável da primeira meta-substituição não pode ocorrer livremente no corpo do termo introduzido pela segunda meta-substituição. Mais formalmente, isto pode ser escrito como:

Lema da Substituição. *[28]*
$M[x := N][y := P] = M[y := P][x := N[y := P]]$, *para todo* x *e* y *tal que* $x \neq y$ *e* $x \notin FV(P)$

O Lema da Substituição demonstra uma propriedade do Cálculo Lambda que as variantes equivalentes devem também apresentar. Inclusive, o cálculo com notação de De Bruijn. A tradução do Lema da Substituição não acontece de forma direta. Uma vez que as variáveis dessa representação tem comportamento dinâmico. Considera-se isto e estabelece-se o seguinte lema:

Lema da Substituição com Índices de De Bruijn. *[29].*
$M[i := N][j := P] = M[j + 1 := P][i := N[j - i := P]]$, *para* i *e* j *tal que* $i \leq j$.

Utilizando a própria noção de meta-substituição, essa representação não é valida para o *framework locally nameless*. A representação de De Bruijn não considera as variáveis livres como nominais. Isso incorre em algumas limitações para com a representação localmente sem nome. Isto pode ser facilmente visto no seguinte Contra-Exemplo:

Contra-exemplo. *O Lema da Substituição com Índices de De Bruijn não vale tomando a Definição 3.13 de meta-substituição.*
Tomando $i = 0$, $j = 0$, $M = \lambda(i + 1)$, $N = j$ *e* $P = x$ *variável livre.*

- *O lado esquerdo da operação resolve-se da seguinte forma:*

$$\begin{aligned}\{j \rightsquigarrow x\}\{i \rightsquigarrow j\}\lambda(i+1) &= \{j \rightsquigarrow x\}\lambda(\{i+1 \rightsquigarrow j\}\ i+1)\\ &= \{j \rightsquigarrow x\}\lambda j\\ &= \lambda(\{j+1 \rightsquigarrow x\}\ j)\\ &= \lambda j\end{aligned}$$

- *Já o lado direito resulta em:*

$$\begin{aligned}\{i \rightsquigarrow \{j - i \rightsquigarrow x\}j\}\{j + 1 \rightsquigarrow x\}\lambda(i + 1) &= \{i \rightsquigarrow \{j - i \rightsquigarrow x\}j\}\lambda(\{j + 2 \rightsquigarrow x\}\ i + 1) \\ &= \{i \rightsquigarrow \{j - i \rightsquigarrow x\}j\}\lambda(i + 1) \\ &= \{i \rightsquigarrow x\}\lambda(i + 1) \\ &= \lambda(\{i + 1 \rightsquigarrow x\}i + 1) \\ &= \lambda x\end{aligned}$$

À priori, a composição de meta-substituições na representação *locally nameless* apenas limita a igualdade entre os índices i e j. Como variável nominais estão nesta representação como variáveis livres, o lema torna-se mais permissivo no que tange a captura de variáveis. Isto essencialmente leva a três representações candidatas para a composicionalidade da meta-substituição definida em 3.13.

$$i \neq j \rightarrow \{j \rightsquigarrow P\}\{i \rightsquigarrow N\}M =_? \begin{cases}\{i \rightsquigarrow N\}\{j \rightsquigarrow P\}M \\ \{i \rightsquigarrow \{j \rightsquigarrow P\}N\}\{j \rightsquigarrow P\}M \\ \{i \rightsquigarrow \{j \rightsquigarrow P\}N\}\{j + 1 \rightsquigarrow P\}M\end{cases}$$

O primeiro candidato é a permutação dos substituições. Essa abordagem é simples o suficiente para N e P variáveis livres. No caso em que N e M são termos sem índices soltos, essa composição ainda vale. Essa limitação sobre N implica que a substituição mais externa não tem qualquer efeito sobre N, e vice-versa. Logo, estas podem ser permutadas. A composição falha em um caso mais geral, quando qualquer M ou N possuem índices soltos. Em qualquer caso, um contra-exemplo é fácil.

Os demais candidatos são similares na estratégia de construção. Em ambos, uma meta-substituição é propagada para dentro da outra. A diferença da construção está em como lidar com o índice a ser substituído. Em um caso, apenas mantém-se o índice de substituição, afinal, o alvo da troca ainda supostamente é o mesmo. No outro, vale-se da definição da meta-substituição para termos com substituições explícitas, a qual incrementa-se o índice. Essa última proposição leva em conta que o resultado de uma meta-substituição e de uma substituição explícita são os mesmos e diferem-se apenas no conceito. Acontece que em ambos os casos, a composicionalidade não vale e isto pode ser mostrado por contra-exemplos bem similares.

Tomando $M = \{j \rightsquigarrow x\}\{i \rightsquigarrow j\}\lambda(i+1)$ onde $i \neq j$. M resolve-se da seguinte forma:

$$\begin{aligned} M &= \{j \rightsquigarrow x\}\{i \rightsquigarrow j\}\lambda(i+1) \\ &= \{j \rightsquigarrow x\}\lambda(\{i+1 \rightsquigarrow j\}\ i+1) \\ &= \{j \rightsquigarrow x\}\lambda j \\ &= \lambda(\{j+1 \rightsquigarrow x\}\ j) \\ &= \lambda j \end{aligned}$$

Contra-exemplo. *$M^1 = \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\{j \rightsquigarrow x\}\lambda(i+1)$, onde $i \neq j$, não é igual à M.*

$$\begin{aligned} M^1 &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\{j \rightsquigarrow x\}\lambda(i+1) \\ &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\lambda(\{j+1 \rightsquigarrow x\}\ i+1) \\ &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\lambda(i+1) \\ &= \{i \rightsquigarrow x\}\lambda(i+1) \\ &= \lambda(\{i+1 \rightsquigarrow x\}\ i+1) \\ &= \lambda x \end{aligned}$$

Contra-exemplo. *$M^2 = \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\{j+1 \rightsquigarrow x\}\lambda(i+1)$, onde $i \neq j$, não é igual à M.*

$$\begin{aligned} M^2 &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\{j+1 \rightsquigarrow x\}\lambda(i+1) \\ &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\lambda(\{j+2 \rightsquigarrow x\}\ i+1) \end{aligned}$$

Este passo tem duas possibilidades em cima da substituição $\{j+2 \rightsquigarrow x\}\ i+1$:

j é antecessor de i:

$$\begin{aligned} M^2 &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\lambda x \\ &= \{i \rightsquigarrow x\}\lambda x \\ &= \lambda(\{i+1 \rightsquigarrow x\}\ x) \\ &= \lambda x \end{aligned}$$

e j não é antecessor de i:

$$\begin{aligned} M^2 &= \{i \rightsquigarrow \{j \rightsquigarrow x\}\ j\}\lambda(i+1) \\ &= \{i \rightsquigarrow x\}\lambda(i+1) \\ &= \lambda(\{i+1 \rightsquigarrow x\}\ i+1) \\ &= \lambda x \end{aligned}$$

O Lema da Substituição não vale para a proposta expandida da *locally nameless representation*. Nenhuma das possibilidades de composição das fórmulas garante a substituição mantendo a igualdade do lema. Isso ocorre pela noção de meta-substituição implementada pelo *framework*. Portanto, construções que necessitem da composicionalidade em seu caso geral, precisam avaliar as condições de forma a restringi-las possibilitando o uso de casos específicos do Lema da Substituição ou reimplementar a noção de meta-substituição para torna-la mais abrangente.

Capítulo 5

Conclusão e Trabalhos Futuros

Este trabalho explora as noções de confluência para o cálculo λ_x na representação do *framework locally nameless* expandido. Toda uma construção é feita a partir da motivação da expansão da gramática pelo Cálculo Lambda com Substituições Explícitas, passando pela demonstração da propriedade da confluência por K. Nakazawa [7], até a motivação para a verificação da prova via assistente de provas Coq e uma tentativa de executar esta tarefa. Certamente o trabalho seria muito mais interessante se nesta conclusão estivesse sendo falado que esta tentativa foi concluída e que a prova do autor que inspirou esse trabalho de fato vale. Porém as construções apresentadas, a base para as provas e a conclusão quanto a composicionalidade de meta-substituição implementada pelo *framework locally nameless* podem servir de inspiração para quem desejar seguir os passos deste trabalho.

A prova da propriedade da confluência, via Z, proposta não é em si tão complexa a ponto de exigir uma grande maturidade matemática. De fato, as provas são quase todas por indução na estrutura de termos lambda. Porém, transpor toda a teoria para o assistente de provas de forma correta, bem definida e conveniente para a prova, exige sim muita habilidade de quem o faz. E talvez este tenha sido o ponto mais complexo no desenvolvimento deste trabalho.

A conveniência das definições talvez seja o ponto mais importante ao lidar com uma formalização via assistente de provas. Escrever as definições de forma que facilite a prova não só exige o pleno domínio da teoria como também da implementação. Por vezes, neste trabalho, a reavaliação das definições não andou lado a lado com as mudanças na estratégia de prova. Reavaliando mais frequentemente as definições, possivelmente alguns pontos em aberto estariam mais próximos de serem fechados.

Ainda sim, as bases para a conclusão da prova são oferecidas. Acredita-se que a partir desse ponto, com os conhecimentos acumulados e transmitidos aqui, a prova de K. Nakazawa pode ser concluída. Este documento é um recorte do trabalho desenvolvido

e, naturalmente, muita coisa interessante como os raciocínios, as ideias de provas, as discussões em torno disso e as dúvidas ficam de fora.

De trabalho futuros, fica a pendência da prova via Coq - ou outro assistente - da confluência do cálculo λ_x. Sendo este o principal objetivo inalcançado deste trabalho. A prova da composição $P \circ B$ ser Z para β por λ_x por si só já é um trabalho desafiador e interessante. Assim como a conclusão dos demais lemas por este trabalho iniciado.

Outro trabalho interessante é a definição de uma meta-substituição em *locally nameless representation* expandido para a qual o Lema da Substituição vale. Certamente esse resultado tem valor, inclusive para compor este trabalho. Sempre buscando rever as definições e simplificar o que já está feito. A composição disto tudo pode enriquecer ainda mais a teoria em torno do Cálculo Lambda com Substituições Explícitas.

Todos os passos deste trabalho, no que tange a formalização em Coq, podem ser encontrados acessando o repositório https://github.com/nunesgrf/lx-confluence.

Referências

[1] Galloway, Andy, Richard F Paige, NJ Tudor, RA Weaver, I Toyn e J McDermid: *Proof vs testing in the context of safety standards*. Em *24th Digital Avionics Systems Conference*, volume 2, páginas 14–pp. IEEE, 2005. 1

[2] Anand, Abhishek, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau e Matthew Weaver: *Certicoq: A verified compiler for coq*. Em *The third international workshop on Coq for programming languages (CoqPL)*, 2017. 2

[3] Pulte, Christopher, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee e Chung Kil Hur: *Promising-arm/risc-v: a simpler and faster operational concurrency model*. Em *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, páginas 1–15, 2019. 2

[4] Gonthier, Georges: *The four colour theorem: Engineering of a formal proof*. Em *Asian Symposium on Computer Mathematics*, páginas 333–333. Springer, 2007. 2

[5] Hales, Thomas C: *A proof of the kepler conjecture*. Annals of mathematics, 162(3):1065–1185, 2005. 2

[6] Barras, Bruno, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy *et al.*: *The Coq proof assistant reference manual: Version 6.1*. Tese de Doutoramento, Inria, 1997. 2

[7] Nakazawa, Koji e Ken etsu Fujita: *Compositional z: Confluence proofs for permutative conversion*. Studia Logica, 104(6):1205–1224, 2016. 2, 39, 53

[8] Church, Alonzo: *A set of postulates for the foundation of logic*. Annals of mathematics, páginas 346–366, 1932. 2

[9] Bernays, Paul: *Alonzo church. an unsolvable problem of elementary number theory. american journal of mathematics, vol. 58 (1936), pp. 345–363.* The Journal of Symbolic Logic, 1(2):73–74, 1936. 2

[10] Abadi, Martin, Luca Cardelli, P L Curien e J J Lévy: *Explicit substitutions*. Journal of functional programming, 1(4):375–416, 1991. 2

[11] Gentzen, Gerhard: *Untersuchungen über das logische schließen. i.* Mathematische zeitschrift, 35, 1935. 4

[12] Ayala-Rincón, Mauricio e Flávio LC De Moura: *Applied Logic for Computer Scientists: Computational Deduction and Formal Proofs*. Springer, 2017. 4, 23

[13] Church, Alonzo e J Barkley Rosser: *Some properties of conversion*. Transactions of the American Mathematical Society, 39(3):472–482, 1936. 12

[14] Nita, Stefania Loredana e Marius Mihailescu: *Lazy evaluation*. Em *Haskell Quick Syntax Reference*, páginas 153–157. Springer, 2019. 12

[15] Copes, Martín, Nora Szasz e Alvaro Tasistro: *Formalization in constructive type theory of the standardization theorem for the lambda calculus using multiple substitution*. arXiv preprint arXiv:1807.01871, 2018. 12

[16] Lins, Rafael Dueire: *A new formula for the execution of categorical combinators*. Em *International Conference on Automated Deduction*, páginas 89–98. Springer, 1986. 13

[17] Lins, Rafael D: *Partial categorical multi-combinators and church-rosser theorems*. 1992. 13

[18] Rose, Kristoffer Høgsbro: *Explicit cyclic substitutions*. Em *International Workshop on Conditional Term Rewriting Systems*, páginas 36–50. Springer, 1992. 13

[19] Bloo, Roel e Kristoffer H Rose: *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*. Em *In CSN-95: Computer Science in the Netherlands*. Citeseer, 1995. 13

[20] Kesner, Delia: *A theory of explicit substitutions with safe and full composition*. arXiv preprint arXiv:0905.2539, 2009. 13

[21] Ayala-Rincón, Mauricio e Cesar Munoz: *Explicit substitutions and all that*. Relatório Técnico, 2000. 14, 15

[22] Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest e Clifford Stein: *Introduction to algorithms*. MIT press, 2009. 16, 23

[23] Pierce, Benjamin C, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg e Brent Yorgey: *Software foundations*. Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html, 2010. 23

[24] Brassard, Gilles e Paul Bratley: *Fundamentals of algorithmics*. Prentice-Hall, Inc., 1996. 24

[25] Felgenhauer, Bertram, Julian Nagele, Vincent van Oostrom e Christian Sternagel: *The z property*. Archive of Formal Proofs, 2016. 30

[26] Moura, Flávio LC de e Leandro O Rezende: *Confluence via the z property in coq*. 35

[27] Charguéraud, Arthur: *The locally nameless representation*. Journal of automated reasoning, 49(3):363–408, 2012. 40

[28] Berghofer, Stefan e Christian Urban: *A head-to-head comparison of de bruijn indices and names*. Electronic Notes in Theoretical Computer Science, 174(5):53–67, 2007. 49

[29] Nipkow, Tobias: *More church–rosser proofs*. Journal of Automated Reasoning, 26(1):51–66, 2001. 49