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

Formalização em Coq da prova de confluência do cálculo λ_x

Danilo Raposo Freire Caldas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Flávio Leonardo Cavalcanti Moura

Brasília
2023

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

Formalização em Coq da prova de confluência do cálculo λ_x

Danilo Raposo Freire Caldas

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Flávio Leonardo Cavalcanti Moura (Orientador)
CIC/UnB

do Curso de Engenharia da Computação

Brasília, 15 de janeiro de 2023

Dedicatória

Dedico esse trabalho a minha família e amigos pelo apoio ao longo de toda a minha graduação.

Agradecimentos

Agradeço a minha família por sempre ter me dado apoio ao longo de toda a minha graduação e aos professores que tive contato ao longo de todo o curso por terem me dado a base para o meu desenvolvimento. Em especial ao professor Flávio pela ajuda e paciência ao longo do desenvolvimento desse trabalho e ao professor Vander por ter me dado a base em métodos formais ao longo de dois anos de iniciação científica.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

Resumo

Um sistema abstrato de redução (ARS) é um par ordenado (A, R) onde A é um conjunto e R é uma relação de redução sobre esse conjunto [1]. A confluência de um ARS, que expressa o determinismo computacional do sistema, é a propriedade que esse sistema tem se, dado um elemento qualquer que possa ser reduzido a partir do fecho transitivo reflexivo da relação do ARS para dois elementos distintos, então esses dois elementos distintos podem ser reduzidos para algum elemento comum a partir do mesmo fecho transitivo reflexivo. Em outras palavras, um ARS é confluente se qualquer divergência pode ser juntada. Existem diversas técnicas distintas para se provar a confluência de um ARS [2], e a que utilizaremos aqui se baseia no fato de que ela pode ser obtida a partir da existência de algum mapeamento que satisfaça a propriedade Z nesse sistema [1, 3].

O cálculo λ é um sistema formal de computação criado por Alonzo Church que é equivalente às máquinas de Turing e que constitui o fundamento teórico do paradigma de programação funcional [4, 5]. Podemos ver o cálculo λ como um sistema abstrato de redução, onde as suas expressões são os elementos de A, e R é o conjunto contendo as reduções do cálculo. Uma das variações do cálculo λ que também pode ser vista como um ARS é o cálculo λ_x, que é um cálculo com substituição explicita, *i.e.*, uma extensão do cálculo λ cujo operador de substituição é um dos construtores da gramática de termos [6, 7].

Esse trabalho tem como objetivo explicar o processo de formalização via o assistente de provas Coq, que é um provador de teorema altamente utilizado tanto no meio acadêmico quanto na, na indústria confluência do cálculo λ_x, conforme [6], no contexto nominal.

Palavras-chave: Coq, Calculo λ, Confluência

Abstract

An abstract rewriting system (ARS) is an ordered pair (A, R), where A is a set and R is a binary relation on this set. Confluence of an ARS, which expresses the determinism of the computational process, is a property satisfied by the system if, given an arbitrary element that reduces to two different elements in the transitive reflexive closure of the binary relation R, then these two different elements can be reduced to a common element in the reflexive transitive closure of R. In other words, an ARS is confluent if every divergence can be joined. There exists different techniques to prove confluence of an ARS, and here we will use one based on the Z property [1].

The λ-calculus is a computational formal system created by Alonzo Church that is equivalent to the Turing machines and constitutes the functional programming paradigm. One can see the λ-calculus as an abstract reduction system, whose expressions are elements in the set A, and R is the set of its reductions. One of the extensions of the λ-calculus that can also be seen as an ARS is the calculus λ_x, which is a calculus with explicit substitutions, i.e. an extension of the λ-calculus whose substitution operator is one of the constructors of the grammar of terms.

The goal of this work is to explain the process of formalization, via the Coq proof assistant, of the confluence property of the λ_x calculus following [6], using the nominal approach so that we can provide a more general overview of its pros and cons in a non trivial example.

Keywords: Coq, *λcalculus*, confluence

Sumário

1 O Cálculo λ e suas extensões **1**
1.1 A estrutura do Cálculo λ . . . 2
1.1.1 α-equivalência e β-redução . . . 4
1.2 O Cálculo λ_x . . . 5
1.2.1 Sintaxe . . . 6

2 Confluência do Cálculo λ_x **9**
2.1 Confluência . . . 9
2.2 A Propriedade Z . . . 10
2.3 A Propriedade Z Composicional . . . 10
2.4 Confluência do Cálculo λ_x . . . 11

3 Coq **13**
3.1 Assistentes de Provas . . . 13
3.2 Provador de Teoremas Coq . . . 14

4 Formalização da Confluência do Cálculo λ_x **15**
4.1 Modelagem . . . 16
4.2 Resultados obtidos . . . 21

5 Conclusão e Trabalhos Futuros **26**
5.1 Conclusão . . . 26
5.2 Trabalhos Futuros . . . 27

Referências **29**

Capítulo 1

O Cálculo λ e suas extensões

Este capítulo introduz noções a respeito da definição do cálculo λ [8] e suas extensões com substituições explícitas. Em particular, apresentaremos o cálculo λ_x [7] que pode ser visto como a versão mais simples e intuitiva dentre os cálculos com substituições explícitas[9, 10, 11, 12, 13, 7].

O cálculo λ é um sistema formal proposto em 1932 como uma formalização de funções pelo matemático estadunidense Alonzo Church. Esse cálculo se baseia no conceito de funções computáveis, e a partir da aplicação, redução e manipulação dessas funções é possível fazer as computações desejadas. Pouco tempo depois do seu desenvolvimento e de forma independente, o matemático britânico Alan Turing propôs o sistema formal conhecido futuramente como máquina de Turing, e no paper onde a máquina de Turing foi definida, em 1936, foi provado que o seu sistema era equivalente ao sistema do cálculo λ proposto por Church [4, 5].

O cálculo λ é a principal base para o desenvolvimento de diversas tecnologias presentes na computação atual, e dentre essas tecnologias, podemos destacar o paradigma de programação funcional [4]. Esse paradigma se baseia na construção de programas baseados não em uma série de instruções que vão resultar na computação de um resultado, mas na aplicação sucessiva de funções em argumentos, que ao longo de sua aplicação fazem a computação necessária. Esse paradigma vem sendo adotado por linguagens de programação não funcionais, dado o seu poder e conveniência. Dentre as linguagens que o estão adotando, podemos citar o Java, o Javascript e o C++. Além disso, existem linguagens de programação que são totalmente baseadas no paradigma funcional, como, por exemplo, as linguagens Lisp, Haskell e OCaML.

1.1 A estrutura do Cálculo λ

Como comentado anteriormente, o cálculo λ é baseado na noção de fuções como cidadãos de primeira classe (first class citzens). Suas expressões são construídas a partir de uma gramática bastante simples, como podemos ver na definição a seguir:

Definição 1.1.1 *[8] Uma expressão no cálculo λ pode ser construída indutivamente a partir da seguinte gramática:*

$$M := x \mid \lambda x.M \mid MM \tag{1.1}$$

Em que x é uma variável de um conjunto infinito enumerável, $\lambda x.M$ é chamado de uma abstração, onde x é uma variável e M uma expressão, e MM é uma aplicação, onde M é uma expressão.

A abstração na definição anterior é um construtor de função, assim uma expressão da forma $\lambda x.M$ corresponde a uma função que tem parâmetro x e corpo M. A aplicação de uma função $(\lambda x.M)$ a um argumento N é feito via a justaposição da função seguida do argumento, ou seja, $(\lambda x.M)N$. Como veremos a seguir, a forma de avaliar a função $(\lambda x.M)$ no argumento N se dá via a substituição de todas as ocorrências livres da variável x em M por N. Este passo de computação é conhecido como β-redução:

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

As definições a seguir serão utilizadas para fornecermos uma definição precisa da operação de substituição denotada por $M[x := N]$ acima, que como veremos é a operação mais importante do cálculo λ.

Pela definição de uma expressão no cálculo λ conseguimos distinguir quais são os conjuntos de variáveis presentes em uma expressão. No entanto, saber apenas qual é o conjunto de variáveis não suficiente para algumas definições, pois as variáveis em uma expressão podem ser divididas em dois conjuntos: Os das variáveis livres e os das variáveis ligadas.

Definição 1.1.2 *[8] O conjunto de variáveis livres de uma expressão e ($\boldsymbol{FV}(e)$) pode ser definido indutivamente da seguinte forma:*

- $\boldsymbol{FV}(x) = \{x\}$
- $\boldsymbol{FV}(\lambda x.M) = \boldsymbol{FV}(M)$ - $\{x\}$
- $\boldsymbol{FV}(MN) = \boldsymbol{FV}(M) \cup \boldsymbol{FV}(N)$

Definição 1.1.3 *[8] O conjunto de variáveis ligadas de uma expressão e ($\boldsymbol{BV}(e)$) pode ser definido indutivamente da seguinte forma:*

- $\boldsymbol{BV}(x) = \emptyset$
- $\boldsymbol{BV}(\lambda x.M) = \boldsymbol{BV}(M) \cup \{x\}$
- $\boldsymbol{BV}(MN) = \boldsymbol{BV}(M) \cup \boldsymbol{BV}(N)$

Analisando essas definições podemos visualizar que as variáveis ligadas de uma expressão são as variáveis relacionadas às abstrações dentro dessa expressão.

Dizemos que ela é uma meta-operação porque o construtor da substituição não faz parte da gramática do cálculo λ, ou seja, a expressão $M[x := N]$ não é nenhum dos construtores da gramática (1.1). Neste sentido, a expressão $M[x := N]$ apresentada anteriormente denota o resultado de substituir todas as ocorrências livres de x em M por N.

A operação de substituição do cálculo λ também tem mais um detalhe: ela é tal que nenhuma ocorrência de variável livre em N pode ser capturada neste processo. Por exemplo, suponha que queiramos substituir todas as ocorrências livres da variável x no termo $\lambda y.xx$ pela variável livre y. Isto corresponde à seguinte expressão $(\lambda y.xx)[x := y]$. Se simplesmente substituirmos as duas ocorrências de x por y obtemos o termo $\lambda y.yy$ e a variável y que antes da substituição era livre passou a ser ligada depois da substituição. Isto não pode ocorrer porque a semântica do termo foi alterada, como pode ser visto ao fazer a aplicação de um termo nas duas abstrações. Para contornar este problema, a operação de substituição é definida da tal forma que esta captura não ocorre. Como isto é feito? Com o renomeamento de variáveis ligadas. Note que o nome de uma variável ligada é irrelevante: de fato, $\lambda x.x$ e $\lambda y.y$ representam a mesma função, uma vez que a aplicação de um termo n. Considerando estas observações, existem diferentes formas de se definir formalmente a operação de substituição, como por exemplo:

Definição 1.1.4 *[8] A substituição no cálculo λ (denotada por $M[x := N]$) é o ato de trocar todas as instâncias livres da variável x em M pela expressão N, de forma que nenhuma variável livre de N seja capturada. Mais precisamente, podemos definir a substituição como:*

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y \ (y \neq x)$
3. $(\lambda x.M)[x := N] \equiv \lambda x.M$
4. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N]) \ (y \neq x)$

5. $(MO)[x := N] \equiv M[x := N]O[x := N]$

Na definição acima, a captura de variáveis é evitada informalmente ao assumirmos a convenção de variáveis de Barendregt [14]:

> **Convenção de variáveis de Barendregt**: Para qualquer termo M do cálculo λ, as variáveis ligadas são diferentes da variáveis livres. Em outras palavras,
>
> $$\mathbf{BV}(M) \cap \mathbf{FV}(M) = \emptyset, \forall M.$$

No item 4 da definição anterior, ao assumirmos a convenção de variáveis de Barendregt, temos que N não pode conter ocorrências livres da variável y já que y é uma variável ligada do termo em consideração. Do ponto de vista formal, a definição acima não é adequada já que precisamos garantir que a variável y não ocorre livre no termo N. Assim, na formalização, o caso 4 da definição acima é adaptado de forma que a variável ligada y é renomeada para uma nova variável z, que não estava em nenhuma das expressões envolvidas na substituição, antes que a substituição seja propagada para dentro da abstração:

$$(\lambda y.M)[x := N] \equiv \lambda z.(M[y := z][x := N])(y \neq x, \text{ e } z \text{ é variável nova})$$

A adaptação acima é possível porque os termos do cálculo λ são considerados módulo α-equivalência, como detalhado a seguir.

1.1.1 α-equivalência e β-redução

Duas expressões são α-equivalentes no cálculo λ se a diferença entre elas for apenas na escolha dos símbolos utilizados para as variáveis ligadas da expressão. Por exemplo, temos que as expressões $\lambda x.x$ e $\lambda y.y$ são α-equivalentes, enquanto que as expressões $\lambda x.xz$ e $\lambda y.yw$ não são. A α-conversão se dá quando substituímos uma ou mais vezes todas as instâncias de uma variável ligada em uma expressão por outra variável, de forma a não ocorrer a captura de variáveis no processo. Mais precisamente, temos a seguinte:

Definição 1.1.5

$$\frac{}{x =_\alpha x}\ (aeq\_var) \qquad \frac{t_1 =_\alpha t_2}{\lambda x.t_1 =_\alpha \lambda x.t_2}\ (aeq\_abs\_same)$$

$$\frac{x \neq y \qquad x \notin \boldsymbol{FV}(t_2) \qquad t_1 =_\alpha (y\ x)t_2}{\lambda x.t_1 =_\alpha \lambda y.t_2}\ (aeq\_abs\_diff)$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{(t_1\ t_2) =_\alpha (t_1'\ t_2')}\ (aeq\_app)$$

Na definição anterior, o termo $(y\ x)t_2$ que aparece na regra aeq_abs_diff representa o resultado de se permutar todas as ocorrências das variáveis x e y no termo t_2. A teoria relacionada com esta ação de permutação é conhecida como Lógica Nominal[15], e será detalhada no Capítulo 4.

Definição 1.1.6 *[5] No cálculo λ, podemos definir a redução beta ($\rightarrow_\beta$) como a relação entre duas expressões que pode ser especificada da seguinte forma:*

- $(\lambda x.M)N \rightarrow_\beta M[x := N]$
- $M \rightarrow_\beta N \Rightarrow ZM \rightarrow_\beta ZN$, $MZ \rightarrow_\beta NZ$ *e* $\lambda x.M \rightarrow_\beta \lambda x.N$

Ter em mente as definições mostradas até agora sobre o cálculo λ é imprescindível para o entendimento das várias áreas de influência desse cálculo [4]. Uma dessas áreas, como mencionado anteriormente, é o paradigma de programa funcional, onde as computações feitas no programa são resultado de aplicações sucessivas de funções em argumentos [4]. No entanto, o salto lógico entre o cálculo λ e a programação funcional pode ser algo muito abrupto, sendo conveniente uma modelagem que sirva como um passo intermediário entre os dois. Nesse contexto, as extensões do cálculo λ com substituições explicitas correspondem a uma representação intermediária, pelo fato de modelar ao nível de objeto a função de alta ordem que é a operação de substituição [7]. Nesse trabalho, vamos nos aprofundar mais especificamente no cálculo com substituição explícita λ_x[16, 17, 18, 19].

1.2 O Cálculo λ_x

O Cálculo λ_x é uma variação do cálculo λ convencional, mas que possui uma estrutura sintática para a substituição de variáveis. Essa nova possibilidade sintática traz algumas mudanças para as definições apresentadas anteriormente, e também traz uma nova forma

de redução no cálculo [6]. Esse cálculo é um exemplo de cálculo com substituição explicita, que é uma forma de tentar modelar a operação de alta-ordem de substituição de forma sintática. Dessa forma é possível ter um maior entendimento de como funcionam as substituições em modelos mais complexos [7].

1.2.1 Sintaxe

Definição 1.2.1 *[6] Uma expressão no cálculo λ com substituição explícita pode ser construída indutivamente a partir da seguinte regra:*

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

Em que $M\langle x := N\rangle$ denota uma substituição explícita, e que diferente da substituição convencional, possui uma estrutura sintática associada.

A construção do conjunto de variáveis livres e variáveis ligadas de uma expressão segue um esquema parecido com o caso do cálculo λ convencional, diferenciando apenas no novo caso para a substituição explicita.

Definição 1.2.2 *Podemos definir indutivamente o conjunto de variáveis livres de uma expressão e ($\boldsymbol{FV}(e)$) da seguinte forma:*

- $\boldsymbol{FV}(x) = \{x\}$
- $\boldsymbol{FV}(\lambda x.M) = \boldsymbol{FV}(M)$ - $\{x\}$
- $\boldsymbol{FV}(MN) = \boldsymbol{FV}(M) \cup \boldsymbol{FV}(N)$
- $\boldsymbol{FV}(M\langle x := N\rangle) = (\boldsymbol{FV}(M)$ - $\{x\}) \cup \boldsymbol{FV}(N)$

Definição 1.2.3 *O conjunto de variáveis ligadas de uma expressão e ($\boldsymbol{BV}(e)$) pode ser definido indutivamente da seguinte forma:*

- $\boldsymbol{BV}(x) = \emptyset$
- $\boldsymbol{BV}(\lambda x.M) = \boldsymbol{BV}(M) \cup \{x\}$
- $\boldsymbol{BV}(MN) = \boldsymbol{BV}(M) \cup \boldsymbol{BV}(N)$
- $\boldsymbol{BV}(M\langle x := N\rangle) = \boldsymbol{BV}(M) \cup \{x\} \cup \boldsymbol{BV}(N)$

Dessa forma, podemos ver que a variável x na expressão $M\langle x := N\rangle$ é ligada dentro da expressão M.

No cálculo λ_x, existe uma meta-operação de substituição denominada meta substituição e que possui a notação apresentada anteriormente, a saber $M[x := N]$, e é definida de forma semelhante a 1.1.4, mas incluindo o caso da substituição explícita:

Definição 1.2.4 *A meta substituição no cálculo λ_x pode ser definida indutivamente da seguinte maneira:*

- $x[x := N] \equiv N$
- $y[x := N] \equiv y$ *(y diferente de x)*
- $(\lambda x.M)[x := N] \equiv \lambda x.M$
- $(\lambda y.M)[x := N] \equiv \lambda z.(M[y := z][x := N])$*(y diferente de x, e z é variável nova)*
- $(MO)[x := N] \equiv M[x := N]O[x := N]$
- $(M\langle x := N\rangle)[x := O] \equiv M\langle x := N[x := O]\rangle$
- $(M\langle x := N\rangle)[y := O] \equiv (M[x := z][y := O])\langle z := N[y := O]\rangle$ *(y diferente de x, e z é variável nova)*

A definição de α-equivalência no cálculo λ_x pode ser vista nas regras de inferência a seguir:

Definição 1.2.5

$$\frac{}{x =_\alpha x}\ (aeq\_var)$$

$$\frac{t_1 =_\alpha t_2}{\lambda x.t_1 =_\alpha \lambda x.t_2}\ (aeq\_abs\_same)$$

$$\frac{x \neq y \qquad x \notin \boldsymbol{FV}(t_2) \qquad t_1 =_\alpha (y\ x)t_2}{\lambda x.t_1 =_\alpha \lambda y.t_2}\ (aeq\_abs\_diff)$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{(t_1\ t_2) =_\alpha (t_1'\ t_2')}\ (aeq\_app)$$

$$\frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1\langle x := t_2\rangle =_\alpha t_1'\langle x := t_2'\rangle}\ (aeq\_sub\_same)$$

$$\frac{t_2 =_\alpha t_2' \qquad x \neq y \qquad x \notin \boldsymbol{FV}(t_1') \qquad t_1 =_\alpha (y\ x)t_1'}{t_1\ \langle x := t_2\rangle =_\alpha t_1'\langle y := t_2'\rangle}\ (aeq\_sub\_diff)$$

Onde $(y\ x)t$ tem a mesma definição vista na definição de α-equivalência do cálculo λ.

Com a adição da nova possibilidade sintática no cálculo com substituição explícita, temos novos casos de reduções para serem definidos. A ideia é simular a β-redução do cálculo λ via uma regra que dispara o processo de simulação, a saber, a regra β_x, e mais um conjunto de regras que executam a simulação. Este conjunto de regras será denotado por π.

Definição 1.2.6 *[6] As reduções β_x e π são definidas da seguinte forma:*

- $(\lambda x.M)N \to_{\beta_x} M\langle x := N\rangle$
- $\pi : \begin{cases} x\langle x := N\rangle & \to_{var} \quad N \\ y\langle x := N\rangle & \to_{gc} \quad y \\ (\lambda y.M)\langle x := N\rangle & \to_{abs} \quad \lambda y.M\langle x := N\rangle \\ (MO)\langle x := N\rangle & \to_{app} \quad M\langle x := N\rangle\ O\langle x := N\rangle \end{cases}$

Podemos ver que a redução β_x tem a mesma ideia da β-redução do cálculo λ, mas restrita a um termo com a substituição explícita. Enquanto que as regras do conjunto π completam a simulação da operação de substituição do cálculo λ.

Capítulo 2

Confluência do Cálculo λ_x

Este capítulo introduz noções a respeito da definição de confluência, como provar que um sistema é confluente e mostra como provar a confluência para o cálculo λ_x

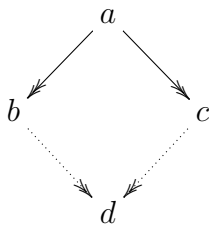
2.1 Confluência

Um sistema abstrato de reescrita (ARS) é um par ordenado (A, R), onde A é um conjunto e R é uma relação de redução sobre esse conjunto. Se $a, b \in A$ então escrevemos $a \to_R b$ para denotar que a está relacionado com b via R. Dada uma relação $\to_R$, podemos definir o seu fecho transitivo reflexivo (representado por $\twoheadrightarrow_R$) indutivamente da seguinte forma [1]:

$$\frac{}{a \twoheadrightarrow_R a}\ (refl) \qquad\qquad \frac{a \to_R b \quad b \twoheadrightarrow_R c}{a \twoheadrightarrow_R c}\ (rtrans)$$

Tendo essas duas definições em mente, podemos definir o que é confluência:

Definição 2.1.1 *[1] Um sistema abstrato de redução (A, R) é dito confluente se para todo a, b e $c \in A$, tais que $a \twoheadrightarrow_R b$ e $a \twoheadrightarrow_R c$ implica a existência de um $d \in A$ onde $b \twoheadrightarrow_R d$ e $c \twoheadrightarrow_R d$. Diagramaticamente, temos a seguinte representação para a noção de confluência*



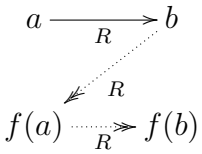
Essa definição traz a ideia de que em um sistema confluente, qualquer divergência na redução a partir de um ponto em comum converge para um ponto em comum.

Como exemplos de sistemas confluentes, podemos citar algumas variações do cálculo λ, como o cálculo $\lambda_{\overline{N}J}$ e o $\lambda\mu_{\overline{N}J}$ [6].

No entanto, a prova de que um sistema é confluente a partir da definição de confluência muitas vezes não é uma tarefa fácil. Por exemplo, a união da redução estrutural com a redução com renomeação traz bastante dificuldade para a prova de confluência do cálculo $\lambda\mu$ [6]. Nesse contexto, a propriedade Z em um ARS pode ser vista como uma solução para o desenvolvimento mais simples da prova de confluência, como será visto a seguir.

2.2 A Propriedade Z

Definimos que uma função $f : A \to A$ satisfaz a propriedade Z se, para todo $a, b \in A$, $a \to_R b$ implica $b \twoheadrightarrow_R f(a) \twoheadrightarrow_R f(b)$. O nome da propriedade se da pelo seguinte diagrama [1]:



Além disso, dizemos que um sistema abstrato de redução satisfaz a propriedade Z se existe uma função f que satisfaça a propriedade Z nesse sistema. Uma das grandes importâncias dessa propriedade é a prova de que ela implica e confluência [20].

Lema 2.2.1 *[1] Para todo sistema abstrato de reescrita (A, R), se existir uma função f tal que f satisfaça a propriedade Z nesse sistema, então (A, R) é confluente.*

Tendo isso em mente, em sistemas onde a prova da confluência pela sua definição é uma prova extremamente complexa, podemos reduzir o nosso esforço para provar a confluência de um sistema ao esforço de provar que existe uma função f que satisfaça a propriedade Z nesse sistema.

2.3 A Propriedade Z Composicional

Em Nakazawa e Fujita [6], é definida uma extensão da noção de Z denominada Z composicional, que nos permite decompor um sistema de reescrita (A,R) em duas partes, digamos (A,R_1) e (A,R_2) tal que $R = R_1 \cup R_2$ para mais facilmente provarmos que o sistema original satisfaz a propriedade Z, e com isso concluirmos que (A, R) é confluente. Para definirmos as propriedades que as funções f_1 e f_2 precisam satisfazer, precisamos primeiro da definição de propriedade Z fraca.

Definição 2.3.1 *[20] Dizemos que uma função f satisfaz a propriedade Z fraca para $\rightarrow$ por $\rightarrow_x$ se, para todo $a, b \in A$, tivermos que $a \rightarrow b$ implica $b \twoheadrightarrow_x f(a)$ e $f(a) \twoheadrightarrow_x f(b)$.*

Tendo essa noção em mente, podemos definir a propriedade Z Composicional

Definição 2.3.2 *[20] Dado um sistema abstrato de redução (A, R), e R sendo uma união das relações $R1$ e $R2$, temos que se existirem funções $f1, f2 : A \rightarrow A$ tais que:*

1. *$f1$ é Z para $\rightarrow_{R1}$*
2. *$a \rightarrow_{R1} b$ implica $f2(a) \twoheadrightarrow_R f2(b)$*
3. *$a \twoheadrightarrow_R f2(a)$ ocorre para todo $a \in Im(f1)$*
4. *$f_2 \circ f_1$ satisfaz a propriedade Z fraca para $R2$ por R*

Então a função $f_2 \circ f_1$ é Z para (A, R).

Além disso, temos outra condição suficiente para a prova de que existe uma função que satisfaz a propriedade Z em (A, R).

Definição 2.3.3 *[20] Dado um sistema abstrato de redução (A, R), e R sendo uma união das relações $R1$ e $R2$, temos que se existirem funções $f1, f2 : A \rightarrow A$ tais que:*

1. *$a \rightarrow_{R1} b$ implica $f1(a) = f1(b)$*
2. *$a \twoheadrightarrow_{R1} f1(a)$ ocorre para todo a*
3. *$a \twoheadrightarrow_R f2(a)$ ocorre para todo $a \in Im(f1)$*
4. *$f_2 \circ f_1$ satisfaz a propriedade Z fraca para $R2$ por R*

Então a função $f_2 \circ f_1$ é Z para (A, R).

Na próxima seção, mostraremos como utilizar a propriedade Z composicional para provar a confluência do cálculo λ_x.

2.4 Confluência do Cálculo λ_x

Uma das formas de provar a confluência do cálculo λ_x é a partir do Lema 2.2.1. Desse forma, achando uma função f que seja Z em λ_x, temos a prova de confluência desejada.

Pela definição 2.3.3 tendo em mente os mapeamentos M^P e M^B a seguir:

- $x^P = x$
- $(\lambda x.M)^P = \lambda x.M^P$
- $(MN)^P = M^P N^P$
- $(M\langle x := N\rangle)^P = M^P[x := N^P]$

- $x^B = x$
- $(\lambda x.M)^B = \lambda x.M^B$
- $((\lambda x.M)N)^B = M^B[x := N^B]$
- $(MN)^B = M^B N^B$ (o.w.)
- $(M\langle x := N\rangle)^B = M^B\langle x := N^B\rangle$

Temos que basta provarmos os seguintes lemas

1. $M \to_\pi N$ implica $M^P = N^P$
2. $M \twoheadrightarrow_\pi M^P$ ocorre para todo M
3. $M \twoheadrightarrow_{\beta_x \cup \pi} M^B$ ocorre para todo $M \in Im(P)$
4. $B \circ P$ satisfaz a propriedade Z fraca para β_x por $\beta_x \cup \pi$

E teremos que $B \circ P$ é Z, provando assim que λ_x é confluente [20].

Capítulo 3

Coq

Este Capítulo introduz informações a respeito do assistente de provas Coq, explicando o seu funcionamento e a sua importância tanto para o desenvolvimento desse trabalho como para o desenvolvimento de tantos outros.

3.1 Assistentes de Provas

Desde o surgimento da ciência da computação, o campo da matemática vem sendo um dos principais pilares para o seu desenvolvimento como área de estudo. Utilizando de campos como a lógica, a álgebra, a teoria dos números e tantos outros, a ciência da computação foi crescendo e nos últimos anos pode ser considerada um dos campos que mais vem se desenvolvendo e se destacando. Felizmente, as contribuições entre a matemática e a ciência da computação não são unilaterais, e várias das descobertas desenvolvidas pela ciência da computação vem contribuindo para o desenvolvimento da matemática. Dentre essas descobertas, podemos destacar os assistentes de provas, que permitem o desenvolvimento e a prova de teoremas matemáticos direto de um computador, usufruindo dos benefícios do mesmo [21].

Um assistente de provas é um sistema computacional que permite tanto a modelagem de sistemas matemáticos e computacionais a partir de uma linguagem formal, permitindo a descrição de estruturas, funções e propriedades do sistema, como também é um sistema que disponibiliza ferramentas para prova de lemas e teoremas sobre o sistema modelado escolhido, permitindo que provas de sistemas matemáticos e computacionais possam ser feitas com o auxílio de um computador [22].

Apesar do desenvolvimento de uma prova em um assistente de provas muitas vezes ser mais demorado e trabalhoso do que o desenvolvimento da mesma em papel e caneta, dado o esforço para fazer a modelagem do sistema e o rigor que a linguagem formal do assistente traz para o desenvolvimento da prova, provas em um assistente tem a grande

vantagem de possuírem um rigor baseado no seu assistente provas que faz com que sejam mais facilmente aceitas do que provas em papel e caneta [22]. Enquanto algumas provas no papel feitas na matemática possam não ser aceitas se forem provas extremamente grandes e complexas, dado o fato de provas muito grandes e com o seu desenvolvimento gerando muitos casos de análise serem mais suscetíveis a erros, provas feitas em um assistente estão baseadas em um sistema amplamente testado e validado por uma grande comunidade, além de que o desenvolvimento baseado em uma linguagem formal faz com que cada passo na prova e na modelagem de um teorema seja checado e validado, garantindo uma maior robustez no desenvolvimento [21]. Por exemplo, podemos citar a prova em Coq do teorema das 4 cores, que anteriormente era considerada controversa entre alguns matemáticos, dado a abundância de configurações que eram necessárias serem analisadas, mas que ao ser feita no provador de teorema Coq trouxe uma maior robustez em seu desenvolvimento, já que todos os passos e computações da prova são checados e validados pelo assistente de provas, garantindo que o desenvolvimento de algo sempre seja feito a partir da prova de algo já validado [21].

3.2 Provador de Teoremas Coq

O Coq é um sistema formal de gerenciamento de provas que permite a definição de algoritmos, sistemas matemáticos, provas e teoremas, além de possuir um ambiente para a escrita de provas validadas passo a passo pelo seu verificador formal. Ele também possui ferramentas para a geração de códigos em linguagens como Haskell e Scheme que possuem certificação de propriedades específicas que foram modeladas a partir da linguagem do Coq [23].

Esse assistente de provas implementa uma linguagem de alto nível chamada Gallinna, que é baseada no cálculo de construção indutiva, trazendo consigo tanto aspectos de lógica de alta ordem quanto aspectos de uma linguagem de programação ricamente tipada [23].

Dentre as várias áreas de utilização do Coq, podemos citar algumas que possuem bastante destaque, como a modelagem de linguagens de programação, onde é possível fazer o estudo de linguagens complexas e permite a verificação de propriedades das linguagens como a segurança, a criação de softwares e hardwares que possuem a propriedades específicas formalmente provadas, tentando aumentar ao máximo a corretude e segurança do produto gerado, e a área acadêmica, onde a modelagem e prova de teoremas utilizando o Coq traz mais segurança e robustez na veracidade das provas [21, 23].

Capítulo 4

Formalização da Confluência do Cálculo λ_x

Esse capítulo explica como foi feita a formalização no assistente de provas Coq da confluência do cálculo λ_x. Vai ser explicada a modelagem feita para os principais aspectos da prova e os principais resultados necessários para provar a confluência, além de algumas provas auxiliares que ou foram muito trabalhosos, ou foram muito utilizados ao logo de todos os lemas subsequentes, e por isso merecem um destaque.

Para fazermos a modelagem e as provas dos lemas propostos nesse trabalho, foi preciso o uso da biblioteca do Coq chamada de `Metalib`[1]. Essa biblioteca é um trabalho do grupo de linguagem de programação da Universidade da Pensilvânia e consiste da junção de várias outras bibliotecas feitas para auxiliar a mecanização no assistente Coq de metateorias de linguagens de programação[2]. Além disso, a modelagem proposta nesse trabalho toma como base uma modelagem do cálculo λ presente na Metalib, e que pode ser achada no seguinte repositório: https://github.com/plclub/metalib/blob/master/Stlc/Nominal.v.

O repositório onde essa formalização foi feita pode ser encontrado em https://github.com/flaviodemoura/lx_confl/tree/danilo

Os arquivos mais importantes presentes nesse repositório são o arquivo `lambda_x.v`, onde está a modelagem de todas as definições feitas, tirando as definições dos mapeamentos P e B, e que também possui a prova de vários lemas auxiliares, o arquivo `lx_confl.v`, onde está a definição dos mapeamentos P e B, e também onde estão as provas de confluência dos lemas descritos em [6] e dos lemas que serão descritos nesse artigo, e o arquivo `ZtoConfl.v`, onde existem alguns lemas sobre a prova de que um ARS é confluente quando ele satisfaz a propriedade Z.

[1] https://github.com/plclub/metalib
[2] https://www.cis.upenn.edu/~plclub/metalib/

4.1 Modelagem

Para iniciarmos a modelagem do cálculo λ_x, é preciso inicialmente ser feita a apresentação da gramática que nos permite construir os termos (ou expressões nominais, daí o nome *n_sexp*) do cálculo:

```
Inductive n_sexp : Set :=
 | n_var (x:atom)
 | n_abs (x:atom) (t:n_sexp)
 | n_app (t1:n_sexp) (t2:n_sexp)
 | n_sub (t1:n_sexp) (x:atom) (t2:n_sexp).
```

Na gramática acima, o tipo *atom* representa objetos sem nenhuma estrutura associada chamados de átomos, mas que a igualdade é decidível, ou seja, dados dois átomos sabemos sempre responder se eles são iguais ou diferentes entre si. Os construtores *n_var*, *n_abs*, *n_app* e *n_sub* representam, respectivamente, variáveis, abstrações, aplicações e substituições explícitas.

Ao definirmos as expressões no cálculo λ_x, temos o necessário para definir recursivamente o conjunto $fv\_nom(t)$ das variáveis livres de t:

```
  Fixpoint fv_nom (n : n_sexp) : atoms :=
 match n with
 | n_var x ⇒ {{x}}
 | n_abs x n ⇒ remove x (fv_nom n)
 | n_app t1 t2 ⇒ fv_nom t1 `union` fv_nom t2
 | n_sub t1 x t2 ⇒ (remove x (fv_nom t1)) `union` fv_nom t2
 end.
```

O predicado *pure* caracteriza os termos que não possuem substituições explícitas:

```
Inductive pure : n_sexp → Prop :=
 | pure_var : ∀ x, pure (n_var x)
 | pure_app : ∀ e1 e2, pure e1 → pure e2 → pure (n_app e1 e2)
 | pure_abs : ∀ x e1, pure e1 → pure (n_abs x e1).
```

Após ser definido o conjunto de variáveis livres de uma expressão, a próxima definição é a de α-equivalência, pois a partir dela é feita a relação de equivalência entre expressões na modelagem do nosso cálculo. No entanto, algumas definições auxiliares como o swap de variáveis em uma expressão, que é a troca de todas as instâncias de uma variável em uma expressão (tanto variáveis livres quanto ligadas) por outra variável precisam ser apresentadas primeiro.

```
Definition swap_var (x:atom) (y:atom) (z:atom) :=
  if (z == x) then y else if (z == y) then x else z.
```

Em seguida, a operação de troca (swap) é estendida para a estrutura dos termos:

```
 Fixpoint swap (x:atom) (y:atom) (t:n_sexp) : n_sexp :=
match t with
| n_var z ⇒ n_var (swap_var x y z)
| n_abs z t1 ⇒ n_abs (swap_var x y z) (swap x y t1)
| n_app t1 t2 ⇒ n_app (swap x y t1) (swap x y t2)
| n_sub t1 z t2 ⇒ n_sub (swap x y t1) (swap_var x y z) (swap x y t2)
```

A definição de α-equivalência pode ser vista na definição indutiva a seguir (c.f. Definição 1.1.5):

```
  Inductive aeq : n_sexp → n_sexp → Prop :=
| aeq_var : ∀ x,
    aeq (n_var x) (n_var x)
| aeq_abs_same : ∀ x t1 t2,
    aeq t1 t2 → aeq (n_abs x t1) (n_abs x t2)
| aeq_abs_diff : ∀ x y t1 t2,
    x ≠ y → x `notin` fv_nom t2 →
    aeq t1 (swap y x t2) →
    aeq (n_abs x t1) (n_abs y t2)
| aeq_app : ∀ t1 t2 t1' t2',
    aeq t1 t1' → aeq t2 t2' →
    aeq (n_app t1 t2) (n_app t1' t2')
| aeq_sub_same : ∀ t1 t2 t1' t2' x,
    aeq t1 t1' → aeq t2 t2' →
    aeq (n_sub t1 x t2) (n_sub t1' x t2')
| aeq_sub_diff : ∀ t1 t2 t1' t2' x y,
    aeq t2 t2' → x ≠ y → x `notin` fv_nom t1' →
    aeq t1 (swap y x t1') →
    aeq (n_sub t1 x t2) (n_sub t1' y t2').
```

Na definição de α-equivalência mostrada acima, os únicos casos que se destacam são os casos da abstração e da substituição explícita onde as variáveis são diferentes. Nesses casos temos que a variável ligada na primeira expressão não pode estar no conjunto de variáveis livres da segunda expressão. Essa definição está de acordo com a ideia de α-equivalência do cálculo λ, pois ao garantirmos isso, temos que as duas expressões vão

possuir o mesmo conjunto de variáveis livres e cada variável livre vai estar na mesma posição nas duas expressões, diferenciando assim as duas expressões apenas pelo conjunto de variáveis ligadas.

Com a definição de α-equivalência, temos o necessário para modelarmos a meta substituição no cálculo λ_x que denotaremos por m_subst. Como esta definição recursiva não é estruturalmente recursiva, utilizaremos as definições auxiliares de tamanho de um termo ($size$) e de substituição limitada a n chamadas recursivas ($subst\_rec$) :

```
Fixpoint size (t : n_sexp) : nat :=
  match t with
  | n_var x ⇒ 1
  | n_abs x t ⇒ 1 + size t
  | n_app t1 t2 ⇒ 1 + size t1 + size t2
  | n_sub t1 x t2 ⇒ 1 + size t1 + size t2
  end.


  Fixpoint subst_rec (n:nat) (t:n_sexp) (u :n_sexp) (x:atom) : n_sexp :=
  match n with
  | 0 ⇒ t
  | S m ⇒ match t with
           | n_var y ⇒
             if (x == y) then u else t
           | n_abs y t1 ⇒
             if (x == y) then t
             else
               let (z,_) :=
                    atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in
                   n_abs z (subst_rec m (swap y z t1) u x)
           | n_app t1 t2 ⇒
             n_app (subst_rec m t1 u x) (subst_rec m t2 u x)
           | n_sub t1 y t2 ⇒
             if (x == y) then n_sub t1 y (subst_rec m t2 u x)
             else
               let (z,_) :=
                    atom_fresh (fv_nom u `union` fv_nom t `union` {{x}}) in
                   n_sub (subst_rec m (swap y z t1) u x) z (subst_rec m t2 u x)
            end
  end.
```

Por fim, a definição de meta substituição (*m_subst*) é limitada a um número de chamadas recursivas iguais ao tamanho do termo onde se quer fazer a substituição:

```
Definition m_subst (u : n_sexp) (x:atom) (t:n_sexp) := subst_rec (size t) t u x.
```

Vale destacar na definição de *subst_rec* no Coq o swap de variáveis nos casos da abstração e da substituição explícita. Isso ocorre pois não temos como garantir que ocorrerá a captura de variável ou não ao fazer a substituição. Dessa forma, o swap por uma variável nova que não estava no conjunto de variáveis livres de nenhuma das expressões envolvidas na substituição garante que não ocorrerá captura ao fazer a substituição.

Por fim, vamos modelar as reduções no cálculo λ_x e as funções P (recursivamente substitui meta substituições por substituições explícitas) e B (*complete development*) que serão utilizados para provar a confluência. Com relação às conversões, fizemos a definição da conversão β_x, da conversão π, da conversão β do cálculo λ original e da conversão β_π, que é a união da conversão π e da β_x.

```
Inductive betax : n_sexp → n_sexp → Prop :=
 | step_betax : ∀ (e1 e2: n_sexp) (x: atom),
      betax (n_app (n_abs x e1) e2) (n_sub e1 x e2).

Inductive pix : n_sexp → n_sexp → Prop :=
| step_var : ∀ (e: n_sexp) (y: atom),
     pix (n_sub (n_var y) y e) e
| step_gc : ∀ (e: n_sexp) (x y: atom),
     x ≠ y → pix (n_sub (n_var x) y e) (n_var x)
| step_abs1 : ∀ (e1 e2: n_sexp) (y : atom),
     pix (n_sub (n_abs y e1) y e2) (n_abs y e1)
| step_abs2 : ∀ (e1 e2: n_sexp) (x y: atom),
     x ≠ y → x `notin` fv_nom e2 →
     pix (n_sub (n_abs x e1) y e2) (n_abs x (n_sub e1 y e2))
| step_abs3 : ∀ (e1 e2: n_sexp) (x y z: atom),
     x ≠ y → z ≠ y → x `in` fv_nom e2 → z `notin` fv_nom e1 → z `notin` fv_nom e2
     → pix (n_sub (n_abs x e1) y e2) (n_abs z (n_sub (swap x z e1) y e2))
| step_app : ∀ (e1 e2 e3: n_sexp) (y: atom),
     pix (n_sub (n_app e1 e2) y e3) (n_app (n_sub e1 y e3) (n_sub e2 y e3)).

Inductive betapi: n_sexp → n_sexp → Prop :=
| b_rule : ∀ t u, betax t u → betapi t u
| x_rule : ∀ t u, pix t u → betapi t u.
```

```
Inductive beta : n_sexp → n_sexp → Prop :=
| step_beta : ∀ (e1 e2: n_sexp) (x: atom),
    beta (n_app (n_abs x e1) e2) (m_subst e2 x e1).
```

Para a definição das reduções, fizemos a definição do fecho contextual reflexivo de uma conversão, que é basicamente a definição de redução feita anteriormente no cálculo λ_x.

```
Inductive ctx (R : n_sexp → n_sexp → Prop): n_sexp → n_sexp → Prop :=
 | step_aeq: ∀ e1 e2, aeq e1 e2 → ctx R e1 e2
 | step_redex: ∀ (e1 e2 e3 e4: n_sexp), aeq e1 e2 → R e2 e3 → aeq e3 e4 →
    ctx R e1 e4
 | step_abs_in: ∀ (e e': n_sexp) (x: atom), ctx R e e' → ctx R (n_abs x e) (n_abs x e')
 | step_app_left: ∀ (e1 e1' e2: n_sexp) , ctx R e1 e1' →
    ctx R (n_app e1 e2) (n_app e1' e2)
 | step_app_right: ∀ (e1 e2 e2': n_sexp) , ctx R e2 e2' → ctx R (n_app e1 e2) (n_app
e1 e2')
 | step_sub_left: ∀ (e1 e1' e2: n_sexp) (x : atom) , ctx R e1 e1' → ctx R (n_sub e1 x
e2) (n_sub e1' x e2)
 | step_sub_right: ∀ (e1 e2 e2': n_sexp) (x:atom), ctx R e2 e2' → ctx R (n_sub e1 x
e2) (n_sub e1 x e2').
```

A definição dos mapeamentos P e B seguem as definições anteriormente vistas.

```
Fixpoint B (t : n_sexp) := match t with
                           | n_var x ⇒ n_var x
                           | n_abs x t1 ⇒ n_abs x (B t1)
                           | n_app t1 t2 ⇒ match t1 with
                                           | n_abs x t3 ⇒
                                                m_subst (B t2) x (B t3)
                                           | _ ⇒ n_app (B t1) (B t2)
                                           end
                           | n_sub t1 x t2 ⇒ n_sub (B t1) x (B t2)
                           end.

Fixpoint P (t : n_sexp) := match t with
                           | n_var x ⇒ n_var x
                           | n_abs x t1 ⇒ n_abs x (P t1)
                           | n_app t1 t2 ⇒ n_app (P t1) (P t2)
                           | n_sub t1 x t2 ⇒ m_subst (P t2) x (P t1)
                           end.
```

4.2 Resultados obtidos

Com relação à estratégia da modelagem do cálculo λ_x, a abordagem seguida foi a definição de uma série de lemas e teoremas relacionados a cada uma das definições descritas anteriormente, com o objetivo de provar propriedades relacionadas às definições que pudessem ser necessárias futuramente, e depois a prova dos lemas descritos em [6]. Ao longo da prova de cada um desses lemas, foi preciso tanto a adição e prova de lemas auxiliares quanto a adequação de lemas já conhecidos à modelagem feita no assistente de provas Coq (Por exemplo, os lemas **refltrans_pure_beta** e **pi_P**, presentes no arquivo `lx_confl.v`).

Infelizmente alguns poucos lemas não foram finalizados, dado problemas tanto da forma como a modelagem no Coq foi feita, mais especificamente o fato de que as variáveis livres geradas automaticamente na meta substituição não podiam ser escolhidas, de forma que muitos problemas que poderiam ser mais facilmente solucionados pela escolha inteligente de uma variável em comum na conversão α se tornavam problemas bastante complexos pelo uso de variáveis geradas automaticamente, quanto por um fator de tempo, mas todos os lemas não finalizados ou são bastante intuitivos de se ver que são corretos ou possuem sua prova em papel já finalizada (Por exemplo, os lemas **aeq_swap_m_subst**, **refltrans_m_subst1** e **refltrans_m_subst2**, presentes no arquivo `lx_confl.v`).

Os lemas desenvolvidos ao longo da modelagem dos termos mencionados anteriormente foram consideravelmente mais fáceis do que os lemas feitos para tentar provar os lemas presentes em [6]. A maioria desses lemas estão relacionados a manipulação e a propriedades tanto de expressões quanto do conjunto de variáveis livres dessas expressões, o swap dessas expressões e a α-equivalência entre elas. Como destaque de uso pode ser pontuada a prova dos Lemas **subst_fresh_eq** e **aeq_swap0** (arquivo `lambda_x.v`), que provam que se uma variável não estiver no conjunto de variáveis livres de uma expressão t, então a meta substituição dessa variável por outra expressão em t é α-equivalente a t, e a prova que se as variáveis x e y não estão na expressão t, então t é α-equivalente ao swap de x e y em t, respectivamente.

Outras provas que se destacaram ao serem amplamente utilizadas ao longo tanto dos lemas principais presentes em [6] quanto nos lemas auxiliares feitos ao longo do trabalho foram as provas que relacionavam a ausência de uma variável no conjunto de variáveis livres de uma expressão e a ausência de uma variável no conjunto de variáveis livres do swap dessa expressão.

Com relação ao desenvolvimento das provas necessárias para provar a confluência do cálculo λ_x, o primeiro lema que se destacou foi o Lema `aeq_swap_m_subst`[3] sobre a distributividade do swap na meta substituição. Esse lema é a base para a maioria das nossas

[3] arquivo `lx_confl.v`

provas envolvendo a α-equivalência e a meta substituição, pois da forma como é implementada a troca de variável visando evitar a captura de variável, não é possível fazermos a redução α desejada, de forma que provas com múltiplas meta substituições acarretam swaps de meta substituições que precisam ser distribuídos para serem solucionados. Infelizmente essa prova não conseguiu ser concluída, mas as partes que faltam da prova são relativamente pequenas em relação ao resto dela.

Vale pontuar que a forma como a meta substituição troca a variável ligada visando evitar a captura de variável foi o maior empecilho no desenvolvimento das provas. Isso se dá pelo fato de, como a variável que é substituída é escolhida de forma automática pelo Coq, não é possível fazer a escolha da mesma variável quando se tem múltiplas meta substituições (salvo em algumas exceções, como o lema **aeq_m_subst_1** do arquivo `lambda_x.v`). Dessa forma, ao se ter que comparar as variáveis das metas substituições para saberem se são iguais ou diferentes, acabamos gerando uma árvore de possibilidades que cresce exponencialmente com o número de meta substituições, tornando as provas cada maiores e mais trabalhosas.

Alguns exemplos de lemas onde a utilização de várias meta substituições causou um grande esforço na resolução da prova foram as provas relacionadas a α-equivalência entre meta substituições, que foram de extrema importância para a resolução das provas, mas que demandaram um grande esforço para serem resolvidas. Como exemplo dessas provas, podemos citar os Lemas **aeq_m_subst_2**, **aeq_m_subst_3** e **aeq_double_m_subst**, presentes no arquivo `lx_confl.v` e que foram essenciais para o desenvolvimento tanto direto como de lemas auxiliares para as provas equivalentes aos Lemas 5.3, 5.4 e 5.5 de [6]. Outros lemas que foram bastante importantes para o desenvolvimento das provas presentes em [6] foram os relacionados ao fecho transitivo reflexivo de uma redução qualquer (**refltrans_abs**, **refltrans_composition3**, **refltrans_app3** e **refltrans_sub3**, presentes em `lx_confl.v`). A partir deles foi possível fazer as provas de todos os lemas a frente que envolviam o fecho transitivo reflexivo de uma das reduções do cálculo λ_x. Infelizmente não foi possível fazer algumas das provas que envolviam redução entre meta substituições (**refltrans_m_subst1** e **refltrans_m_subst2**, presentes em `lx_confl.v`), mas os casos análogos para todos os tipos de expressões foram resolvidas, de forma que a validade do lema é intuitiva.

O primeiro lema presente em [6] que foi provado foi o Lema 5.3.1.

Lema 4.2.1 *$M \to_\pi N$ implica $M^P = N^P$.*

Ou seja, a redução por π de uma expressão para outra expressão implica na igualdade dos mapeamentos P das duas expressões. Para se adequar a nossa modelagem no Coq, a prova fala que a redução implica na α-equivalência entre as duas expressões. Para resolver essa prova, foi feita indução em $\to_\pi$.

Seguindo, os próximos lemas provados foram os lemas remanescentes do Lema 5.3.

Lema 4.2.2 *M^P é puro.*

Lema 4.2.3 *M puro implica $M^P = M$.*

Lema 4.2.4 *M puro implica $M\langle x := N\rangle \twoheadrightarrow_\pi M[x := N]$.*

O Lema 5.3.2 (Lema 4.2.2 acima) diz que qualquer expressão no conjunto imagem do mapeamento P é puro, e esse lema pode ser facilmente provado pela indução na expressão. O Lema 5.3.3 (Lema 4.2.3 acima) diz que se uma expressão é pura, então a aplicação de P nela produz ela mesma, e esse lema foi resolvido facilmente utilizando as mesmas ideias do Lema 5.3.2. Por fim, o último lema provado do Lema 5.3 foi o 5.3.4 (Lema 4.2.4 acima), que diz que para toda expressão $e1$ pura, nos temos que a substituição explícita de $e1$ por uma variável e outra expressão qualquer pode ser reduzido pelo fecho transitivo reflexivo da relação π para a meta substituição de $e1$ pela mesma variável e pela mesma outra expressão. A prova desse lema se dá pela indução em $e1$ e foi relativamente tranquila após a finalização das provas relativas a meta substituições.

O próximo lema provado presente em [6] foi o Lema 5.4. Nele, são provados lemas relativos à redução β do cálculo λ original. Para fazer a adaptação desses lemas para a nossa modelagem, foi utilizado a nossa modelagem do $\rightarrow_\beta$ e os lemas foram provados para expressões puras, uma vez que o conjunto de expressões puras do cálculo λ_x é igual ao conjunto de expressões do cálculo λ.

Lema 4.2.5 *$M \rightarrow N$ ocorrer em λ_x implica $M^P \twoheadrightarrow_\beta N^P$ no cálculo λ.*

Lema 4.2.6 *$M \rightarrow_\beta N$ ocorrer em λ implica $M \twoheadrightarrow N$ em λ_x.*

O primeiro lema é o Lema 5.4.1 (Lema 4.2.5 acima), que diz que se uma expressão reduz a outra no cálculo λ_x, então a aplicação de P na primeira expressão reduz pelo fecho transitivo reflexivo de β a aplicação de P na segunda expressão. Como foi provado que qualquer expressão na imagem de P é pura, então a aplicação P nas duas expressões está em λ, e assim eles podem ser reduzidos por $\twoheadrightarrow_\beta$. A prova desse lema se dá pela indução em $\rightarrow$ e foi relativamente extensa, mas tranquila. O segundo lema é o 5.4.2 (Lema 4.2.6 acima), que diz que, para duas expressões puras, se uma delas pode ser reduzida por β para a outra, então elas também podem ser reduzidas pelo fecho reflexivo transitivo de β_π em λ_x, onde β_π é a união de β_x e π. Para a resolução dessa prova, foi feita a indução em $\rightarrow_\beta$.

Com a prova do Lema 5.4, é possível fazer a prova de que o mapeamento B é Z em relação ao cálculo λ de acordo com [6]. Infelizmente a prova desse lema não foi concluída,

uma vez que esse foi um dos últimos lemas feitos, e o pouco tempo para a sua resolução somado a uma complexidade de prova que não permitiu a sua finalização nas primeiras tentativas resultou na falha em finalizar a prova. Deixamos esse lema para trabalhos futuros.

Por fim, a última prova feita foi a prova do Teorema 5.5, que é a prova dos requisitos necessários para provar que a função $B \circ P$ é Z de acordo com a prova Z composicional em [6].

Lema 4.2.7 *Para provar a confluência de λ_x, é suficiente provar os seguintes pontos:*

- $M \to_\pi N$ *implica* $M^P = N^P$.
- $M \twoheadrightarrow_\pi M^P$ *ocorre para todo* M.
- $M \twoheadrightarrow M^B$ *ocorre para todo* M *puro.*
- $M \to_{\beta_x} N$ *implica* $N \twoheadrightarrow M^{PB} \twoheadrightarrow N^{PB}$.

A estrutura geral desta prova é como a seguir:

1. A primeira prova a ser feita é a prova de que se uma expressão reduz para outra por π, então a aplicação de P nas duas expressões são iguais, e como esse é o enunciado do Lema 5.3.1, então ela já tinha sido feita.
2. A segunda prova foi a prova de que qualquer expressão pode ser reduzida pelo fecho transitivo reflexivo de π para a sua imagem em P, e ele é facilmente provada pela indução na expressão.
3. A terceira prova é a prova de que qualquer termo puro é reduzido pelo fecho transitivo reflexivo de $\beta\pi$ à sua imagem em B, e essa também foi facilmente resolvida utilizando a indução na expressão junto com os lemas provados anteriormente.
4. Por último, temos a prova de que dado duas expressões M e N do cálculo λ_x, temos que $M \to_{\beta_x} N$ implica $N \twoheadrightarrow M^{PB} \twoheadrightarrow N^{PB}$. Para resolver essa prova, foi feita a divisão dela na prova de $N \twoheadrightarrow M^{PB}$ e $M^{PB} \twoheadrightarrow N^{PB}$. A primeira prova pode ser feita a partir da indução em $\to_{\beta_x}$, com o caso mais complicado necessitando a prova de que $M^B[x := N^B] \twoheadrightarrow (M[x := N])^B$. Infelizmente essa prova não foi possível de ser finalizada. A segunda parte da última prova foi feita a partir das provas do Lema 5.4 e com a ajuda da prova de que B é Z para β.

Dessa forma, temos a prova de que $B \circ P$ é Z, e por consequência temos que λ_x é confluente [6].

Capítulo 5

Conclusão e Trabalhos Futuros

5.1 Conclusão

Este trabalho apresentou o desenvolvimento no assistente de provas Coq da prova de confluência do cálculo λ_x via a propriedade Z Composicional, prova essa que foi feita seguindo as estratégias e os lemas propostos em [6]. A abordagem utilizada é baseada na lógica nominal[15] que tem se mostrado um formalismo interessante para a formalização de teorias que possuem *binders*[24]. No caso da formalização do cálculo λ e suas variantes, a definição da noção de α-equivalência exige alguns cuidados que possam evitar, por exemplo, o uso excessivo de trocas de átomos (*swaps*). Neste trabalho, utilizamos a biblioteca `Metalib` que possui um ferramental que nos deu um bom suporte no desenvolvimento das provas. No entanto, a noção de α-equivalência disponível nesta biblioteca faz uso do *swap* de uma forma sistemática, o que resultou em provas particularmente longas. Por exemplo, a prova do lema `aeq_m_subst_3` (arquivo `lx_confl.v`) que estabelece a α-equivalência é parcialmente compatível com a operação de meta substituição possui mais de 1700 linhas!

Infelizmente, por causa problemas muitas vezes relacionados a modelagem no assistente de provas, nem todas as provas foram concluídas, mas todas as provas pendentes ou são bastante intuitivas ou foram provadas no papel em [6]. Essas provas serão deixadas para trabalhos futuros.

Com a conclusão desse trabalho, conseguimos verificar a veracidade dos lemas descritos em [6] e trazer uma robustez maior à esses lemas. Além disso, produzimos um material que pode ser utilizado futuramente no desenvolvimento de outras provas no Coq que precisem da prova de confluência do cálculo λ_x.

5.2 Trabalhos Futuros

Dado o fato da prova de confluência do cálculo λ_x não ter sido totalmente concluída, uma vez que alguns dos lemas auxiliares necessários para a finalização da prova não foram finalizados, temos a possibilidade de trabalhos futuros com o objetivo de concluir e possivelmente complementar a nossa prova.

O primeiro lema auxiliar que não foi provado é relativo à distributividade do swap na meta substituição, que tem o nome de *aeq_swap_m_subst* no arquivo `lx_confl.v`. Nesse lema, temos que um swap sendo aplicado a uma meta substituição é α-equivalente a uma meta substituição onde o swap anterior está sendo aplicado a cada um dos termos dessa substituição, ou seja, o swap pode ser propagado para dentro da meta substituição.

$$\text{swap } x\ y\ (M[z := N]) =_\alpha (\text{swap } x\ y\ M)[(\text{swap } x\ y\ z) := (\text{swap } x\ y\ N)]$$

O segundo lema cuja prova está pendente é a proposição 1.4.1 em [25], conhecido como Lema da Substituição, que versa sobre a uma espécie de distributividade da meta substituição quando uma condição específica é atendida, e essa proposição é a parte que falta para provar a parte d do Lema 5.5 presente em [6]. No arquivo `lx_confl.v` do Coq esse lema recebeu o nome de *m_subst_lemma*.

Também não foram finalizadas algumas provas referentes ao fecho reflexivo transitivo de meta substituições. Mais especificamente, temos que dado duas expressões e o fecho contextual reflexivo de uma relação R ($\rightarrow_{ctx(R))}$), temos as seguintes provas como objetivos:

$$M \twoheadrightarrow_{ctx(R)} N \text{ implica } M[x := Z] \twoheadrightarrow_{ctx(R))} N[x := Z]$$

e

$$M \twoheadrightarrow_{ctx(R)} N \text{ implica } Z[x := M] \twoheadrightarrow_{ctx(R))} Z[x := N]$$

Como já temos provas semelhantes a essa para todos os casos gramaticais do cálculo λ_x, a veracidade do lema acima é intuitiva. No arquivo presente na nossa modelagem do Coq esses lemas foram chamados de $refltrans\_m\_subst1$ e $refltrans\_m\_subst2$.

A quarta parte da prova que ficou de fora foi a prova de que o mapeamento B é Z para β, que teve seu lema nomeado como *Z_property_B_beta* no Coq, e temos que esse lema é verídico de acordo com [6].

Além disso, temos como trabalho futuro fazer algumas adaptações na prova para que a prova de confluência do cálculo λ_x possa ser exportada e utilizada em outras provas do Coq de acordo com a definição 2.3.3 presente em [6]. Essas adaptações são necessárias uma vez que, enquanto na definição 2.3.3 o primeiro ponto indica que a igualdade entre $f1(a)$ e $f1(b)$ é uma igualdade sintática entre os termos, temos que na nossa prova do

Z composicional feita nesse trabalho, a igualdade entre $f1(a)$ e $f1(b)$ é uma igualdade módulo α-equivalência, que é uma igualdade mais abrangente se comparada a igualdade sintática. Dessa forma, é preciso provar que o Lema 5.5 em [6] considerando a igualdade módulo α-equivalência implica no lema considerando a igualdade sintática.

Por último, vale pontuar como possível trabalho futuro a refatoração das provas de forma a torna-las menores e mais legíveis, uma vez que muitos trechos de código estão repetidos e poderiam ser refatorados em mais lemas auxiliares ou substituídos por algumas automações de prova.

Referências

[1] Moura, Flávio L. C. e Leandro O. Rezende: A formalization of the (compositional) z property. Em Fifth Workshop on Formal Mathematics for Mathematicians - FMM21. http://flaviomoura.info/files/fmm21.pdf. v, vi, 9, 10

[2] Terese: Term Rewriting Systems, volume 55 de Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003. v

[3] van Oostrom, Vincent: Z; syntax-free developments. Em Kobayashi, Naoki (editor): 6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021), volume 195 de Leibniz International Proceedings in Informatics (LIPIcs), páginas 24:1–24:22, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, ISBN 978-3-95977-191-7. v

[4] Michaelson, Greg: AN INTRODUCTION TO FUNCTIONAL PROGRAMMING THROUGH LAMBDA CALCULUS. Dover Publications, 2011. v, 1, 5

[5] Barendregt, Henk e Erik Barendsen: Introduction to lambda calculus, 2000. https://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf. v, 1, 5

[6] Nakazawa, Koji e Ken etsu Fujita: Compositional Z: Confluence Proofs for Permutative Conversion. Studia Logica, 104(6):1205–1224, 2016. v, vi, 6, 8, 10, 15, 21, 22, 23, 24, 25, 26, 27, 28

[7] Kesner, D.: The Theory of Calculi with Explicit Substitutions Revisited. Em CSL, páginas 238–252, 2007. v, 1, 5, 6

[8] The lambda calculus. https://plato.stanford.edu/entries/lambda-calculus/. 1, 2, 3

[9] Abadi, M., L. Cardelli, P. L. Curien e J. J. Lévy: Explicit Substitutions. Journal of Functional Programming, 1(4):375–416, 1991. 1

[10] Kamareddine, F. e A. Ríos: Extending a λ-calculus with Explicit Substitution which Preserves Str
Journal of Functional Programming, 7:395–420, 1997. 1

[11] Muñoz, C. A.: Confluence and Preservation of Strong Normalisation in an Explicit Substitutions C
Em Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996, páginas 440–447, 1996. 1

[12] Curien, Pierre Louis, Thérèse Hardin e Jean Jacques Lévy: Confluence properties of weak and strong calculi of explicit substitutions. Journal of the ACM, 43(2):362–397, março 1996, ISSN 0004-5411, 1557-735X. 1

[13] Milner, R.: Local Bigraphs and Confluence: Two Conjectures: (Extended Abstract). ENTCS, 175(3):65–73, 2007. 1

[14] Barendregt, H. P.: The Lambda Calculus : Its Syntax and Semantics (Revised Edition). North Holland, 1984. 4

[15] Pitts, Andrew M.: Nominal logic, a first order theory of names and binding. Information and Computation, 186(2):165–193, novembro 2003, ISSN 08905401. https://linkinghub.elsevier.com/retrieve/pii/S089054010300138X, acesso em 2023-01-15. 5, 26

[16] Lins, R.: Partial categorical multi-combinators and Church Rosser theorems. Relatório Técnico 7/92, Computing Laboratory, University Kent at Canterbury, maio 1992. 5

[17] Lins, R.: A new formula for the execution of categorical combinators. 8th Conference on Automated Deduction (CADE), volume 230 of LNCS:89–98, 1986. 5

[18] Rose, K.: Explicit cyclic substitutions. Em Rusinowitch, Michaël e Jean Luc Rémy (editores): 3rd International Workshop on Conditional Term Rewriting Systems (CTRS), volume 656, páginas 36–50. Springer-Verlag, 1992. 5

[19] Bloo, R. e K. Rose: Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Em CSN-95: COMPUTER SCIENCE IN THE NETHERLANDS, páginas 62–72, 1995. 5

[20] Dehornoy, P e V van Oostrom: Z, proving confluence by monotonic single-step upperbound functions. Logical Models of Reasoning and Computation (LMRC-08), página 85, 2008. 10, 11, 12

[21] Pierce, Benjamin C., Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg e Brent Yorgey: Logical Foundations. Software Foundations series, volume 1. Electronic textbook, maio 2018. Version 5.5. http://www.cis.upenn.edu/ bcpierce/sf. 13, 14

[22] Geuvers, H.: Proof Assistants: History, Ideas and Future. Sadhana, 34(1):3–25, 2009. 13, 14

[23] Team, The Coq Development: The Coq Proof Assistant. Zenodo, outubro 2021. 14

[24] Abel, Andreas, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer e Kathrin Stark: POPLMark reloaded: Mechanizing proofs by logical relations. Journal of Functional Programming, 29:e19, 2019, ISSN 0956-7968, 1469-7653. https://www.cambridge.org/core/product/identifier/S0956796819000170/type/journal_article, acesso em 2022-07-31. 26

[25] Ong, C. H.L.: Lambda calculus. https://www.irif.fr/~faggian/LMFI_2021/biblio/Ong_Lambda_Calculus_Notes.pdf. 27