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

Sobre a Confluência do Cálculo λ_x na Representação Nominal

Caio Albuquerque Brandão

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

Orientador
Prof. Dr. Flavio Leonardo Cavalcanti Moura

Brasília
2021

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

Sobre a Confluência do Cálculo λ_x na Representação Nominal

Caio Albuquerque Brandão

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

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

Prof.a Dr.a Daniele Nantes Sobrinho
MAT/UnB

Prof. Dr. Daniel Lima Ventura
INF/UFG

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

Brasília, 09 de novembro de 2021

Dedicatória

A todos que lutam

Agradecimentos

Agradeço ao meu orientador Flávio Leonardo Cavalcanti de Moura por todos os conhecimentos passados, pela paciência e atenção em todo esse trabalho.

Agradeço aos meus pais Maria Jacinta Almeida Albuquerque e Armando Henrique Silva Brandão pelo amor, carinho, compreensão e apoio.

Agradeço a todos os meus colegas da maratona de programação pelo apoio, colaboração e amizade ao longo do curso.

Agradeço a todos que me incentivaram de alguma forma a seguir em frente.

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

O cálculo lambda é um modelo teórico do início da década de 1930 e é a base do paradigma de programação funcional utilizado por linguagens como *Ocaml*, *Haskell* e *Lisp*. Com o desenvolvimento de *software* assistentes de prova ao longo dos anos, diversas formas de representação do cálculo lambda em ambiente formal foram propostas, dentre elas a que mais se assemelha à representação em papel e lápis é a representação nominal. A principal operação do cálculo lambda, o conceito da β-redução, que associa uma aplicação de uma abstração a uma operação de substituição, pode ser entendida como um passo de computação. Com a necessidade de um formalismo que trouxesse a operação de substituição ao nível da linguagem objeto foram criados os cálculos com substituições explícitas, dentre eles o abordado por esse trabalho, o cálculo λ_x. A confluência é uma propriedade que afirma que qualquer divergência converge. No cálculo λ_x essas reduções são representadas pela regra B e pelo conjunto de regras x. Uma forma de provar que o cálculo lambda é confluente é através da propriedade Z composicional. Nesse trabalho introduzimos o cálculo lambda e algumas das suas representações em assistentes de prova, depois introduzimos o *software* usado nesse trabalho, o Coq, e depois passamos pelas contribuições desse trabalho para a infraestrutura da prova da confluência de λ_x.

Palavras-chave: Coq, cálculo lambda, λ_x, substituições explícitas

Abstract

The lambda calculus is a computational model from the 1930s and it is the base of the functional programming paradigm used in languages such as *Ocaml*, *Haskell* and *Lisp*. Through the years with the development of proof assistant software, several ways of representing the lambda calculus were proposed, among them the one which is the most similar with paper and pencil reasoning is the nominal representation. The main operation of lambda calculus, the concept of β-reduction, which associates an application of an abstraction with a substitution operation can be understood as a computational step. In the need of a formalism that brings the substitution to the object level, the lambda calculi with explicit substitution were created, among them the one which was considered in this project, λ_x. Confluence is a property that says all divergence can be joined. In λ_x, the reduction steps are the rules B and the set of rules x. A way to prove that the lambda calculus is confluent is using the compositional Z property. In this project we introduce the lambda calculus and some of its representations in proof assistant *software*, then we introduce the *software* used in this project, Coq, and then we talk about some of the contributions of this project to the infrastructure of the proof of confluence of λ_x.

Keywords: Coq, lambda calculus, λ_x, explicit substitutions

Sumário

Introdução

Alonzo Church em meados da década de 1930 desenvolveu um modelo teórico que tinha como objetivo criar um conjunto de postulados para representar as fundações da lógica matemática [1]. Apesar de não ter dado continuidade ao projeto, a teoria funcional de seu trabalho foi aprimorada e deu origem ao que se chama hoje de cálculo lambda.

O cálculo lambda puro é composto de duas operações básicas, a **aplicação** e a **abstração**. Outra operação do cálculo lambda puro é a **substituição**, essa operação ocorre sem captura de variável e é definida a nível de metalinguagem. Através de um passo chamado β-redução associamos uma aplicação de uma abstração a uma substituição. A β-redução, no modelo teórico, pode ser entendida como um passo de computação que pode ser aplicado em um termo lambda.

A confluência no cálculo lambda é uma propriedade que diz que qualquer divergência na redução converge. Uma das formas de provar a confluência do cálculo lambda é usando a propriedade Z composicional como mostrada no trabalho de Nakazawa e Fujita [2].

Com a necessidade de definir a substituição ao nível de objeto, foram propostos diversos cálculos com substituições explícitas [3, 4, 5, 6], dentre eles provavelmente o mais simples é o λ_x. A ideia desse trabalho é contribuir na construção da prova formal da confluência do cálculo λ_x via a propriedade Z composicional em Coq utilizando o trabalho de Moura e Rezende [7]. Para definir λ_x em Coq foi escolhida a representação nominal [8].

Dentre as formas de se representar o cálculo lambda e suas extensões no ambiente formal dos assistentes de prova, o mais parecido com a representação clássica de papel e lápis é a representação nominal. Nela todas as variáveis são representadas por um nome, chamados de átomos, que são usados para representar tanto as variáveis livres quanto as ligadas.

Para conclusão da prova é necessário definir e provar um conjunto de lemas auxiliares, para isso foi necessárion definir um conjunto de lemas, teoremas e propriedades relativos à α-equivalência. A ideia desse trabalho é utilizar as operações de conjunto da biblioteca *Metalib* para poder definir as variáveis livres de um termo lambda e a partir daí montar

toda a estrutura do cálculo lambda usando como base o trabalho do arquivo Nominal.v do trabalho de *DeepSpec summer school* 2017 [8].

No primeiro capítulo abordaremos um histórico do cálculo lambda bem como sua definição, estrutura, regras e importância. Falaremos sobre cálculos com substituições explícitas e sua motivação e concluiremos comparando as formas de representação do cálculo lambda no ambiente formal de prova.

No segundo capítulo daremos uma breve introdução ao assistente de prova Coq utilizado nesse trabalho e abordaremos um exemplo apresentando o passo a passo de uma prova em Coq. Falaremos um pouco sobre provas por indução em Coq e apresentaremos um exemplo de uma prova por indução.

No terceiro capítulo apresentaremos como o cálculo foi definido em Coq, suas regras básicas e lemas auxiliares que foram provados nesse trabalho. Por fim apresentaremos mais detalhes sobre a propriedade Z e como foi definida a base da prova da confluência.

Na conclusão comentaremos as dificuldades do trabalho apresentado bem como expectativas para trabalhos futuros na conclusão da prova da confluência. Abordaremos ao longo desse documento os trabalhos que serviram como base da definição do cálculo lambda e que forneceram uma gama de lemas e teoremas que foram fundamentais na execução das provas.

O repositório desse trabalho pode ser encontrado em `https://github.com/flaviodemoura/lx_confl/`.

Capítulo 1

Cálculo Lambda com substituições explícitas e suas representações

Este capítulo introduz o cálculo Lambda com substituições explícitas bem como diversas formas de representação que são utilizadas em assistentes de prova.

1.1 Cálculo Lambda

O cálculo lambda é um modelo teórico de computação na lógica matemática desenvolvido por Alonzo Church em meados da década de 1930 [1]. O projeto original tinha como objetivo propor um conjunto de postulados para compor os fundamentos da lógica formal, porém foi mostrado ser inconsistente por Kleene e Ross em 1935 [9]. A teoria funcional desenvolvida no trabalho inicial, porém, foi continuada, mostrou-se consistente em [10] e evoluiu para o que hoje é chamado cálculo lambda, um modelo computacional completo, equivalente à Máquina de Turing, em termos de computabilidade, e que é a base do paradigma de programação funcional.

O cálculo lambda é composto de duas operações básicas, a aplicação e a abstração. A aplicação pode ser entendida pelo conceito básico de função, dado dois termos N e M , a aplicação de N em M é denotada por NM. A abstração, por sua vez, é uma operação que associa uma variável ao argumento de uma função, denotamos a abstração de uma variável x em um termo N como $\lambda x.N$. A abstração pode ser entendida como uma função complementar da aplicação, o termo $\lambda x.N$, por exemplo, seria equivalente a uma função com parâmetro x, e corpo N.

Na teoria do cálculo lambda, diferencia-se variáveis que são ou não afetadas pela abstração em um termo. Chamamos de variáveis livres as que não são ligadas a um abstrator, e variáveis ligadas as que são ligadas a um abstrator.

Assim, denominamos VL o conjunto de variáveis livres indutivamente definido por:

- $\mathrm{VL}(x) = \{x\}$, para qualquer variável x;
- $\mathrm{VL}(MN) = \mathrm{VL}(M) \cup \mathrm{VL}(N)$;
- $\mathrm{VL}(\lambda x.M) = \mathrm{VL}(M) \backslash \{x\}$.

A variável x no termo $\lambda x.M$ é dita ligada [11].

Um termo lambda pode sofrer uma operação de troca de variáveis chamada *swap*. Denotamos um *swap* das variáveis x por y em M como $(xy)M$. Esta operação pode ser definida pelas seguintes regras de inferência:

$$\frac{}{(x\ y)x = y}\ (swap\_var1)$$

$$\frac{}{(x\ y)y = x}\ (swap\_var2)$$

$$\frac{z \notin \{x, y\}}{(x\ y)z = z}\ (swap\_var3)$$

$$\frac{(x\ y)z = w \qquad (x\ y)t = t'}{(x\ y)\lambda z.t = \lambda w.t'}\ (swap\_abs)$$

$$\frac{(x\ y)t_1 = t_1' \qquad (x\ y)t_2 = t_2'}{(x\ y)(t_1\ t_2) = t_1'\ t_2'}\ (swap\_app)$$

Outra operação importante é o renomeamento de variáveis ligadas ou α-conversão em um termo lambda. Essa operação é importante porque termos que diferem apenas pelo nome das suas variáveis ligadas são equivalentes em termos computacionais.

Assim, termos lambda que diferem apenas no nome de suas variáveis ligadas são ditos α-equivalentes. Mais precisamente, a α-equivalência é definida pelas regras de inferência a seguir:

$$\frac{}{x \equiv_\alpha x}\ (aeq\_var)$$

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

$$\frac{t_1 \equiv_\alpha (xy)t_2}{\lambda x.t_1 \equiv_\alpha \lambda y.t_2}\ (aeq\_abs\_diff) \qquad x \neq y, x \notin VL(t_2)$$

$$\frac{t_1 \equiv_\alpha t'_1 \qquad t_2 \equiv_\alpha t'_2}{t_1 t_2 \equiv_\alpha t'_1 t'_2}\ (aeq\_app)$$

Uma das dificuldades encontradas nesse trabalho foi justamente provar propriedades que envolvem a manipulação da operação de *swap*. Muitos resultados auxiliares tiveram que ser definidos e provados e muitas vezes essa manipulação não era trivial e demandava vários passos, o que resultou em um trabalho de certa forma bastante verboso.

Denotamos por $M\{x/N\}$, o resultado da substituição de todas as ocorrências livres de x por N em M evitando a captura de variáveis, definida indutivamente por:

- $x\{x/N\} = N$
- $y\{x/N\} = y \quad (x \neq y)$
- $(M_1 M_2)\{x/N\} = M_1\{x/N\}\ M_2\{x/N\}$
- $(\lambda x.M)\{x/N\} = \lambda x.M$
- $(\lambda y.M)\{x/N\} = \lambda y.(M\{x/N\}) \quad (x \neq y)$ e $y \notin \mathrm{VL}(N)$

Note que no último caso, se $y \in \mathrm{VL}(N)$ então é necessário garantir que y não ocorre livre em N para evitar a captura.

Por fim, definimos a β-redução como sendo a regra que retorna o resultado de aplicar uma função $\lambda x.M$ a um argumento N:

$$((\lambda x.M)N) \rightarrow_\beta M\{x/N\}$$

A β-redução é a principal operação do cálculo lambda e representa a noção de um passo de computação.

1.2 Substituições explícitas

Apesar de fazer parte da definição de β-redução, a operação de substituição é definida no nível de metalinguagem, ou seja, é definida informalmente e fora do escopo que compõe a base do cálculo. Em contextos formais como os assistentes de prova, a substituição precisa ser definida explicitamente, ou seja, no nível da linguagem objeto. Chamamos as extenções do cálculo lambda com essa característica de cálculos com substituições explícitas. Um exemplo é o proposto por Abadi *et al* (1991)[12], nele as substituições estão no nível de objeto, ou seja, são incorporadas nas regras da linguagem.

Existem diversas formas de estender o cálculo lambda com substituições explícitas. A forma mais simples é provavelmente o λ_x [3, 4, 5, 6]. A especificação de λ_x é dada por um sistema de reduções onde os termos, assim como no cálculo lambda são considerados módulo α-equivalência, ou seja, módulo renomeamento de variáveis ligadas [13].

A seguir, apresentamos o sistema de regras do cálculo λ_x:

$$(\lambda x.t)u \to t[x/u]$$
$$x[x/u] \to u$$
$$y[x/u] \to y, (x \neq y)$$
$$(t_1 t_2)[x/u] \to (t_1[x/u]t_2[x/u])$$
$$(\lambda y.v)[x/u] \to \lambda y.v[x/u]$$

Onde a primeira regra é denominada regra B e as demais regras que fazem parte do cálculo de substituições explícitas associado denominadas x.

As regras de λ_x foram as usadas nesse trabalho e correspondem ao comportamento mínimo que pode ser encontrado na maioria dos cálculos com substituições explícitas na literatura [13]. Esse trabalho concentrou-se na prova formal da confluência de λ_x com representação nominal.

1.3 Representações

Existem diversas formas para a representação dos termos lambda, sendo elas divididas em duas abordagens: abordagens concretas, onde as variáveis são representadas por nomes ou índices; e abordagens onde os termos são representados como funções de metalinguagem. Dentre as representações concretas as principais abordagens quanto ao nome dos termos temos, a abordagem nominal, os índices de De Bruijn e as representações local com e sem nome. [14]

1.3.1 Nominal

Na representação nominal cada variável é definida por um nome, ou seja, por um identificador próprio. Essa é a representação mais intuitiva, a que mais se aproxima da representação escrita em papel do cálculo, e foi a escolhida desse trabalho. Nela nós trabalhamos com a α-conversão assim como na representação escrita, isso não ocorre nas outras representações porque existe a separação entre duas classes de variáveis, livres e ligadas. Por exemplo, o termo $\lambda x.x$ é diferente sintaticamente do termo $\lambda y.y$, mas eles são α-equivalentes, ou seja, diferem apenas pelo nome das variáveis ligadas [15].

1.3.2 Notação de De Bruijn

Na notação de De Bruijn, cada variável é representada por um índice, ou seja, são representadas por números naturais. Cada termo possui índices que representam variáveis de acordo com a sua "distância" para o abstrator que a liga [16]. A maior vantagem trazida pelos índices de De Bruijn é que agora o renomeamento de variáveis não é mais necessário, ou seja, os termos α-equivalentes agora têm representação única.

Por exemplo, o termo $\lambda x.(\lambda y.xy)xy$, é formado por duas abstrações e as variáveis x e y são ligadas ao primeiro e segundo corpos abstratores respectivamente. Na representação de De Bruijn esse termo assumiria a forma $\lambda(\lambda 10)01$, no contexto contendo a variável (livre) y como primeiro elemento. Assim, nesta notação precisamos de um contexto (lista ou vetor de variáveis) para permitir a indexação das variáveis livres do termo. Note que é possível que uma variável assuma diferentes índices de acordo com a posição no termo em que se encontra e isso algumas vezes pode não ser intuitivo para o leitor.

Outra desvantagem se dá pelos índices serem passíveis de mudanças dentro do termo em que aparecem ao longo do processo de redução, necessitando de operações como a de *shifting* que consiste em incrementar todos os índices de um termo depois de um limite dado, isso pode deixar os lemas de uma prova desnecessariamente complexos [15] e de difícil legibilidade. Outro problema do cálculo lambda representado com índices de De Bruijn é a dificuldade em traduzir provas informais para o ambiente de prova, como exposto em [17] na prova do lema da substituição.

1.3.3 Notação com nomes locais

Para diferenciar variáveis livres e ligadas e evitar captura, foi proposta a representação com nome local [15]. Aqui já não é necessária uma condição a mais para evitar a captura em substituições. Porém ainda há o problema que a α-conversão ainda deve ser tratada de

modo explícito para relacionar termos α-equivalentes, ou seja, ainda é necessário utilizar o *swap*.

A gramática dessa representação pode ser descrita como:

$$t \ := \text{bvar}\, x \mid \text{fvar}\, p \mid \text{abs}\, x\, t \mid \text{app}\, t\, t$$

1.3.4 Notação sem nomes locais

Na representação local sem nome, as variáveis ligadas são representadas por índices de De Bruijn e as variáveis livres são representadas por um nome. Essa representação evita de ter que lidar explicitamente com a α-conversão e com as operações de *shifting* apesar de ainda haver a separação das variáveis em duas classes distintas.

A gramática da notação sem nomes locais se assemelha muito à da notação local sem nomes, a diferença é que a variável ligada agora é representada por um índice:

$$t \ := \text{bvar}\, i \mid \text{fvar}\, p \mid \text{abs}\, x\, t \mid \text{app}\, t\, t$$

Por exemplo, o termo $\lambda x.xy$ que contém a variável ligada x e a variável livre y é representado como $\lambda.0y$. Note que é necessário que o índice faça referência a um abstrator válido, o termo $\lambda.1$ por exemplo não é válido pois temos apenas um abstrator [14].

Capítulo 2

O Assistente de provas Coq

Este capítulo apresenta detalhes do ambiente de formalização, da representação nominal do cálculo lambda, estratégias de implementação da prova no assistente de prova Coq e lemas propostos.

2.1 Ambiente

Foi utilizado o assistente de provas Coq para realização do trabalho. O Coq provê uma linguagem formal para escrever definições formais, algoritmos executáveis e teoremas junto com um ambiente de desenvolvimento de provas verificadas por máquinas. [18]

O Coq é muito útil para definições indutivas, inclusive os termos definidos como base nesse trabalho são todos indutivos. Com isso, nas provas em geral podemos "quebrar" os termos nas definições que o compõe. Um exemplo do que pode ser definido indutivamente são os números naturais.

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

Essa representação dos naturais utilizada no Coq, presente no primeiro capítulo de [19], representa os números naturais na chamada representação unária. A letra O maiúscula representa o número 0 e o construtor S representa a função sucessor.

A partir daí podemos definir funções e suas propriedades a partir de definições como essa dos números naturais. Como um assistente de prova, o Coq provê um ambiente controlado para a formalização desses teoremas.

2.1.1 Um exemplo de prova em Coq

Em Coq, as provas são constituídas de instruções chamadas de táticas, introduzidas pelo usuário, que corresnpondem aos passos da prova [18]. As táticas são aplicadas levando em consideração o objetivo da prova e as hipóteses apresentadas. Algumas táticas são básicas da linguagem, outras são definidas em bibliotecas que podem ser importadas no código.

Um exemplo de prova feita em Coq é a da reflexividade da diferença de átomos. Aqui o símbolo '<>' representa o sinal de diferença $\neq$:

```
Lemma neq_symmetric : forall (x:atom) (y:atom)

    x <> y -> y <> x.
```

As variáveis x e y aqui são do tipo **atom** que são componentes básicos em uma definição indutiva. Podemos escrever o enunciado dessa propriedade como sendo

Sejam x e y átomos. Se $x \neq y$ então $y \neq x$.

Vamos apresentar a prova desse lema nos próximos parágrafos, utilizando uma sequência de quadros onde à direita estão as táticas utilizadas e à esquerda o estado da prova depois de aplicada a tática. Vale lembrar que esta não é a única forma de provar tal lema e que o caminho escolhido para a prova tem objetivo didático.

No ambiente de prova separamos as hipóteses do objetivo, assim nosso primeiro passo será separar a definição das variáveis e a expressão à esquerda da implicação do objetivo da prova. Para isso utilizamos a tática intros que corresponde ao passo de prova que elimina o *forall*, ou seja, toma as variáveis como arbitrárias.

intros x y H.	- x, y : atom - H : x <> y ========== y <> x

Para prosseguir na prova devemos entender como foi definida a relação de diferença no ambiente de prova. A diferença pode ser entendida como a negação da igualdade e em Coq, representamos uma negação como uma implicação à falsidade. Em Coq escrevemos $x \neq y$ como not $(x = y)$. Assim utilizamos a tática unfold para renomearmos a negação e devemos fazer isso tanto na hipótese H quanto no objetivo da prova. Para isso utilizamos o caractere '*' para deixar explícito que esse renomeamento deve acontecer em todas as incidências de not na prova.

unfold not in *.	- x, y : atom - H : x = y -> False ========== y = x -> False

Prosseguindo, mais uma vez introduzimos o lado esquerdo da implicação ($y = x$) como hipótese utilizando intros H0.

intros H0.	- x, y : atom - H : x = y -> False - H0 : y = x ========== False

Agora como temos a falsidade no objetivo então podemos aplicar a hipótese H utilizando a tática apply.

apply H.	- x, y : atom - H : x = y -> False - H0 : y = x ========== x = y

Conseguimos expor a igualdade $x = y$ no objetivo da prova porém a hipótese H0 que relaciona essas duas variáveis diz que $y = x$. Como a igualdade é simétrica essas duas expressões são equivalentes, porém não é possível aplicar diretamente a hipótese, antes devemos utilizar uma tática nativa no Coq chamada symmetry.

symmetry.	- x, y : atom - H : x = y -> False - H0 : y = x ========== y = x

Como agora temos uma hipótese igual ao objetivo, podemos assumir que o nosso objetivo é válido. Para concluir a prova podemos simplesmente utilizar a tática assumption, assim concluímos a prova de neq_symmetric.

2.2 Provas por indução

A indução é uma estratégia utilizada para provar a validade de uma proposição para um número infinito de termos. Em uma indução simples sempre provamos o passo inicial e

depois provamos para os demais casos.

A prova por indução funciona muito bem para provar propriedades de conjuntos enumeráveis como o dos números naturais. Um exemplo simples de prova por indução é a que diz que a soma dos n primeiros números ímpares é igual a n^2:

$$1 + 3 + 5 + ... + (2n - 1) = n^2$$

Provando o passo inicial, quando $n = 1$:

$$1 = 1^2, \text{ o que é verdade}$$

para provar a proposição para um $n > 1$ qualquer, supomos que a mesma proposição vale para $n - 1$:

$$1 + 3 + 5 + ... + ((2n - 1) - 1) + (2n - 1) = n^2$$
$$(n - 1)^2 + (2n - 1) = n^2$$
$$n^2 - 2n + 1 + 2n - 1 = n^2$$
$$n^2 = n^2$$

Em Coq a indução é facilitada através da tática induction que aplica a indução ao termo passado como parâmetro e já adiciona a hipótese de indução após a prova do caso base.

Uma definição do lema da soma dos n primeiros ímpares está presente em [20]:

```
Fixpoint sum_odd_n (n:nat) : nat :=
match n with 0 => 0 | S p => 1 + 2 * p + sum_odd_n p end.

Lemma sum_odd_n_pow_2: forall n:nat, sum_odd_n n = n*n.
```

Não entraremos em detalhes da prova desse lema nesse trabalho pois envolve a aplicação de alguns lemas auxiliares para ser concluída.

Como um termo lambda é composto por um conjunto finito de construtores, podemos usar a indução para provar propriedades que valem para qualquer termo. Quase todas as provas desse trabalho foram concluídas utilizando indução. Em λ_x os casos que um termo pode assumir são o de uma variável, uma abstração, uma aplicação ou uma substituição explícita, portanto para provar qualquer propriedade que vale para um termo lambda qualquer basta provar que a propriedade vale para cada caso que o termo pode assumir. Nesse trabalho também serão apresentados lemas que foram provados por indução não em um termo, mas em predicados como uma α-equivalência, uma redução no fecho transitivo reflexivo ou até no tamanho do termo.

Capítulo 3

Estruturando a confluência via Z composicional

Nesse capítulo apresentaremos como os termos de λ_x foram definidos em Coq, apresentaremos também as representações de *swap*, da metasubstituição, das regras de redução de λ_x e da α-equivalência no ambiente de prova. Apresentaremos também todos os lemas auxiliares que foram propostos e provados.

3.1 A representação nominal em Coq

A representação nominal é a mais intuitiva dentre as concretas. Em assistentes de prova assim como no papel, cada variável é definida com um nome identificador [14].

Para a definição das operações básicas e dos lemas fundamentais do cálculo, foi utilizado o trabalho do repositório em [8] que utiliza como base as funções da biblioteca *Metalib* [21]. O arquivo Nominal.v contém a definição do cálculo Lambda puro com representação nominal. Parte do nosso trabalho consiste na adição do construtor para substituições explícitas e a adaptação das provas dos teoremas já existentes, além de outras propriedades adicionais.

A biblioteca *Metalib* provê uma série de ferramentas e lemas que foram utilizados na prova como operações entre conjuntos como união e pertinência. Além disso ela fornece a tática default_simp que é uma coleção de "passos" simples que são aplicados ao objetivo da prova várias vezes seguidas. Essa tática simplificou bastante o trabalho em diversas provas.

Os termos no ambiente de prova foram definidos de forma indutiva, porém para representarmos as variáveis foi necessário a notação de conjuntos. Na *Metalib* os elementos que

integram um conjunto são chamados de átomos. Assim, cada variável utilizada no cálculo foi definida como um átomo. Como vantagem dessa representação, não foi necessário criar dois tipos de classe para diferenciar variáveis livres e ligadas, todas são representadas por átomos.

Os primeiros lemas auxiliares que foram definidos e provados como contribuição desse trabalho são relativos a conjuntos.

Lema 1 (notin_singleton_is_false)**.** *x 'notin' (singleton x) -> False.*

Lema 2 (remove_singleton_empty)**.** *remove x (singleton x) [=] empty.*

Lema 3 (remove_singleton)**.** *remove t1 (singleton t1) [=] remove t2 (singleton t2).*

As duas primeiras provas necessitaram de propriedades de conjuntos definidas na *Metalib*. A terceira prova é trivial a partir do Lema 2.

3.2 A estrutura do cálculo λ_x em Coq

3.2.1 Termos básicos

Os termos do cálculo λ_x são definidos indutivamente a partir de construtores para variáveis, abstrações, aplicações e substituições explícitas. No ambiente proposto, um λ_x-termo é denominado *n_sexp* e possui os seguintes construtores:

- n_var x: representa o termo composto apenas por uma variável x.
- n_abs x t: representa o termo composto por uma abstração λx.t.
- n_app t1 t2: representa o termo composto por uma aplicação (t1 t2).
- n_sub t1 x t2: representa o termo composto por uma substituição explícita t1[x/t2].

Para auxiliar a prova da confluência foi definida a propriedade pure. Um termo é dito puro se ele não possui uma substituição explícita no seu corpo, ou seja, corresponde a um termo lambda.

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

Outra propriedade de um termo é o seu tamanho, aqui, essa operação pode ser entendida como o número de vezes que um termo pode sofrer uma redução. Utilizamos o tamanho para definir a metasubstituição mais à frente.

```
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.
```

3.2.2 Variáveis livres

As variáveis livres de um termo formam um conjunto de átomos e as operações ‘in‘ e ‘notin‘ indicam se uma variável pertence ou não ao conjunto. Foi definido o conjunto fv_nom que relaciona cada termo do cálculo λ_x com o seu conjunto de variáveis livres.

Utilizando as operações de conjuntos definidas na *Metalib*, definimos indutivamente *fv_nom* do seguinte modo:

```
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.
```

3.2.3 A operação swap

Como explicado no capítulo anterior, na representação nominal é necessária uma operação de renomeamento para representar a α-equivalência. Para isso foi definida uma operação básica de troca, o swap_var que faz uma associação simples entre variáveis e, em cima

disso, foi montada a estrutura indutiva *swap* que define a permutação de átomos em termos.

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

A partir de swap_var definimos *swap*.

```
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)
  end.
```

O *swap* é uma ferramenta importante para provas com representação nominal, já que sem ela não conseguimos representar a α-equivalência. Em casos de dois termos α-equivalentes com variáveis ligadas diferentes, o *swap* é fundamental para que possamos comparar seus subtermos. Por esse motivo foi necessário definir alguns lemas auxiliares que facilitam a manipulação do *swap* em provas futuras, são eles:

Lema 4 (swap_size_eq)**.** *size (swap x y t) = size t.*

Um termo mantém o seu tamanho caso sofra um *swap*, facilmente provado por indução.

Lema 5 (pure_swap)**.** *pure t -> pure (swap x y t).*

Esse lema é trivial, por indução em t os casos do cálculo lambda puro não deixam de ser puros caso sofram um *swap*, e pela definição, uma substituição explícita não é pura.

Lema 6 (swap_id)**.** *swap x x t = t.*

Lema 7 (swap_symmetric)**.** *swap x y t = swap y x t.*

Essas duas propriedades são básicas de *swap* e foram usadas em quase todas as provas desse trabalho, a exceção são as provas que usaram apenas operações simples. Esses dois lemas são triviais e já estavam presentes no arquivo base Nominal.v na versão pura do cálculo lambda. Ambos podem ser facilmente provados por indução em t.

Lema 8 (swap_symmetric_2)**.** *x <> x' -> y <> y' -> x <> y'-> y <> x' -> swap x y (swap x' y' t) = swap x' y' (swap x y t).*

"Esse lema foi definido para auxiliar na prova do Lema 20, que será apresentado mais à frente no texto, e também pode ser facilmente provado por indução. Aqui podemos dizer que não importa a ordem de uma sequência de trocas de átomos diferentes dois a dois, ele sempre resulta no mesmo termo.

Lema 9 (shuffle_swap)**.** *w <> z -> y <> z -> (swap w y (swap y z t)) = (swap w z (swap w y t)).*

Lema 10 (swap_involutive)**.** *swap x y (swap x y t) = t.*

Lema 11 (swap_equivariance)**.** *swap x y (swap z w t) = swap (swap_var x y z) (swap_var x y w) (swap x y t).*

Lema 12 (fv_nom_swap)**.** *z 'notin' fv_nom t -> y 'notin' fv_nom (swap y z t).*

Esses quatro lemas também já estavam presentes em Nominal.v na versão do cálculo lambda puro e auxiliaram nas provas mais longas deste trabalho. A única prova não trivial dentre as quatro acima é a de swap_equivariance porque exigiu utilizar a reescrita definida no Lema 6.

Lema 13 (fv_nom_swap_2)**.** *z 'notin' fv_nom (swap y z t) -> y 'notin' fv_nom t.*

Esse lema é a volta do Lema 12 e foi definido apenas para auxiliar na prova do Lema 20.

Lema 14 (swap_remove_reduction)**.** *remove x (remove y (fv_nom (swap y x t))) [=] remove x (remove y (fv_nom t)).*

Esse lema foi definido apenas para auxiliar a prova do Lema 17.

Lema 15 (fv_nom_swap_remove)**.** *x <> y -> x <> y0 -> x 'notin' fv_nom (swap y0 y t) -> x 'notin' fv_nom t.*

Esse lema foi definido apenas para auxiliar a prova do Lema 20.

Lema 16 (fv_nom_remove_swap)**.** *x <> y -> x <> y0 -> x 'notin' fv_nom t -> x 'notin' fv_nom (swap y0 y t).*

De modo análogo esse lema é a volta do lema anterior e foi definido para provar o Lema 19 que é a volta do Lema 20.

Lema 17 (remove_fv_swap)**.** *x 'notin' fv_nom t -> remove x (fv_nom (swap y x t)) [=] remove y (fv_nom t).*

Apesar de ter sido definido nesse trabalho como um lema auxiliar apenas para provar o Lema 21, esse lema mostrou-se bastante longo. Diferente da maioria das provas desse trabalho, o caso em que t é uma variável não é trivial, pois quando se remove o único

elemento do conjunto das variáveis livres, obtemos um conjunto vazio e não havia uma igualdade definida para conjuntos vazios na *Metalib*. Para provar esse caso foi necessário usar o Lema 3.

3.2.4 α-Equivalência

A α-equivalência relaciona dois termos que diferem apenas pelo nome das variáveis ligadas como equivalentes. A definição em Coq é dada a seguir:

```
Inductive aeq : n_sexp → n_sexp → Prop :=
  | aeq_var : forall x,
    aeq (n_var x) (n_var x)
  | aeq_abs_same : forall x t1 t2,
    aeq t1 t2 → aeq (n_abs x t1) (n_abs x t2)
  | aeq_abs_diff : forall 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 : forall t1 t2 t1' t2',
    aeq t1 t1' → aeq t2 t2' →
    aeq (n_app t1 t2) (n_app t1' t2')
  | aeq_sub_same : forall 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 : forall 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').
```

A α-equivalência é uma relação de equivalência, além de ser preservada quando ocorre uma troca de variáveis em ambos os lados. Essas propriedades são fundamentais para a prova da confluência de λ_x como será mostrado mais à frente.

Lema 18 (aeq_size)**.** *aeq t1 t2 -> size t1 = size t2.*

É fácil ver que dois termos α-equivalentes tem o mesmo tamanho

Lema 19 (aeq_swap1)**.** *aeq t1 t2 -> aeq (swap x y t1) (swap x y t2).*

Lema 20 (aeq_swap2)**.** *aeq (swap x y t1) (swap x y t2) -> aeq t1 t2.*

A primeira parte da bi-implicação de aeq_swap foi provada por indução em aeq t1 t2. Nessa prova todos os casos foram triviais com exceção dos casos onde a α-equivalência entre t_1 e t_2 possui variáveis ligadas com nomes diferentes. Aqui foi necessário abrir em diversos casos da prova e fazer várias manipulações utilizando os lemas auxiliares. Foram usados os Lemas 6, 9, 7, 8, 10, 11, 12, 16.

A segunda parte foi provada usando indução em t_1 e para cada caso da indução foi feita uma indução em t_2 utilizando *induction*. De modo análogo os casos não triviais foram os que t_1 e t_2 possuíam variáveis ligadas com nomes distintos e foram provados usando uma manipulação parecida com a do lema anterior utilizando os mesmos lemas auxiliares citados.

Lema 21 (aeq_fv_nom)**.** *aeq t1 t2 -> fv_nom t1 [=] fv_nom t2.*

Esse lema foi provado por indução em $t_1 \equiv_\alpha t_2$ e foi bem útil para substituir termos α-equivalentes nos lemas seguintes.

Lema 22 (aeq_sym)**.** *aeq t1 t2 -> aeq t2 t1.*

Foi provada por indução em $(t_1 \equiv_\alpha t_2)$, mais uma vez os casos não triviais foram os que t_1 e t_2 constituiam abstrações ou substituições explícitas com variáveis ligadas diferentes. Foram utilizados os Lemas 7, 10, 12 e 21 para completar a prova.

Lema 23 (aeq_abs)**.** *y `notin` fv_nom t -> aeq (n_abs y (swap x y t)) (n_abs x t).*

Lema 24 (aeq_sub)**.** *y `notin` fv_nom t1 -> aeq (n_sub (swap x y t1) y t2) (n_sub t1 x t2).*

Esses dois lemas retratam um caso especial de quando a variável ligada não está no termo sujeito a abstração e substituição explícita. Não foi necessário utilizar indução nesses lemas, apenas separar em dois casos, $x = y$ e $x \neq y$.

Lema 25 (swap_reduction)**.** *x `notin` fv_nom t -> y `notin` fv_nom t -> aeq (swap x y t) t.*

Lema 26 (aeq_swap_swap)**.** *z 'notin' fv_nom t -> x 'notin' fv_nom t -> aeq (swap z x (swap x y t)) (swap z y t).*

Os lemas acima foram propostos unicamente para auxiliar na prova do Lema 27 e foram provados por indução em t.

Lema 27 (aeq_trans)**.** *aeq t1 t2 -> aeq t2 t3 -> aeq t1 t3.*

O lema aeq_trans foi provado pela indução em t1, e foi necessário utilizar os Lemas 4, 6, 7, 9, 10, 15, 18, 19, 20, 21, 22, 25 e 26.

3.2.5 A Metasubstituição

Denominamos metasubstituição como a substituição em nível de metalinguagem, ou seja, é a mesma substituição do cálculo lambda puro. Um problema encontrado aqui, como no cálculo lambda puro, é como devemos evitar a captura de variável, ou seja, temos que garantir que variáveis livres não sejam capturadas pela metasubstituição. Para isso podemos fazer o renomeamento das variáveis ligadas para uma variável nova no termo a ser inserido. Por exemplo, em $(\lambda x.y)\{y/x\}$, deve haver um renomeamento de variáveis com um átomo novo para evitar a captura. Para isso foi usada a estrutura atom_fresh da biblioteca *Metalib* que justamente define a representação de um átomo novo.

Como no *swap* de abstrações e substituições explícitas há o renomeamento de variáveis ligadas, essa função não é estruturalmente recursiva. Por isso, como parâmetro para essa função, é definido o tamanho termo.

Com isso definiu-se as regras de metasubstituição onde o parâmetro n é um inteiro que é decrementado a cada operação.

```
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 t
          else
            let (z,_) := atom_fresh (fv_nom u ‘union‘ fv_nom t ‘union‘ x ‘union‘
fv_nom t2) in n_sub (subst_rec m (swap y z t1) u x) z (subst_rec m t2 u x)
      end
    end.
```

Portanto a operação de metasubstituição, no ambiente de prova, é definida como a substituição recursiva com tamanho do termo a sofrer a substituição como parâmetro.

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

Foram definidos alguns lemas para manipular a metasubstituição.

Lema 28 (subst_swap_reduction)**.** *(swap x y (m_subst u z t)) = (m_subst (swap x y u) (swap_var x y z) (swap x y t)).*

Lema 29 (aeq_m_subst_1)**.** *aeq t1 t2 -> aeq (m_subst t3 x t1) (m_subst t3 x t2).*

Lema 30 (aeq_m_subst_2)**.** *aeq t1 t2 -> aeq (m_subst t1 x t3) (m_subst t2 x t3).*

Lema 31 (aeq_m_subst_same)**.** *aeq t1 t1' -> aeq t2 t2' -> aeq (m_subst t1 x t2) (m_subst t1' x t2').*

Lema 32 (aeq_subst_diff)**.** *x <> y -> aeq t1 t1' -> aeq t2 (swap x y t2') -> aeq (m_subst t1 x t2) (m_subst t1' y t2').*

A prova desses lemas não foi concluída, um dos motivos é que em alguns casos é necessário manipular operações de *swap* dentro de uma metasubstituição e não está claro como isso pode ser feito já que não existe nenhum lema auxiliar que relaciona um *swap* com o termo a ser substituído em uma metasubstituição.

3.2.6 Regras de redução

O cálculo λ_x pode ser visto como uma extensão do cálculo lambda onde a β-redução é simulada em diversas etapas a partir da regra B que dispara esta simulação, seguida do conjunto de regras x que completam esta simulação. Assim, um passo do cálculo associado x é uma operação que aplica uma redução em termos que possuem substituições explícitas. No corpo da prova, as regras B e x são agrupadas em uma estrutura denominada betapi que representa o sistema λ_x apresentado na Seção 1.2. No ambiente de prova desse trabalho, porém, por seguir a nomenclatura de Nakazawa e Fujita[2] denominamos a regra B de betax, e as regras x de pix.

```
Inductive betax : n_sexp -> n_sexp -> Prop :=
  | step_betax : forall (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 : forall (e: n_sexp) (y: atom),
    pix (n_sub (n_var y) y e) e
  | step_gc : forall (e: n_sexp) (x y: atom),
    x <> y -> pix (n_sub (n_var x) y e) (n_var x)
  | step_abs1 : forall (e1 e2: n_sexp) (y : atom),
    pix (n_sub (n_abs y e1) y e2) (n_abs y e1)
  | step_abs2 : forall (e1 e2: n_sexp) (x y: atom),
    x <> y -> pix (n_sub (n_abs x e1) y e2) (n_abs x (n_sub e1 y e2))
  | step_app : forall (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 : forall t u, betax t u -> betapi t u
  | x_rule : forall t u, pix t u -> betapi t u.
```

Para o ambiente de prova, porém, apenas essas definições não são suficientes para aplicar uma redução em qualquer parte do termo. Por exemplo em $\lambda y.((\lambda x.t)t')$ não podemos

aplicar diretamente uma β-redução direta para obter $\lambda y.t\{x/t'\}$, isso ocorre porque o termo se encontra dentro de uma abstração e a regra betax só pode ser aplicada no termo que é absolutamente compatível com a regra. Para isto, definimos o **fecho contextual** reflexivo, módulo α-equivalência, que permite aplicar as regras de redução em qualquer posição do termo. No ambiente de prova, a definição é aplicada a uma regra genérica **R** que relaciona dois termos, mas na prática na prova da confluência de λ_x R é uma redução betapi.

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

Por fim para representarmos uma redução qualquer do cálculo como uma redução betapi aplicada no fecho contextual, foi definido lx dessa forma:

```
Definition lx t u := ctx betapi t u.
```

3.3 Confluência

Antes de entrar na prova da confluência vamos apresentar algumas definições necessárias para prosseguir.

Sistema de reescrita

Seja M um conjunto e $\rightarrow$ uma relação binária sobre M . Um **sistema de reescrita** é um par (M, $\rightarrow$) onde $\rightarrow$ é denominada a **relação de redução** ou **relação de reescrita** correspondente ao sistema.

Notação. Seja R = (M, $\rightarrow$) um sistema de reescrita. Denotamos por:

- $\leftarrow$ a relação inversa de $\rightarrow$, de forma que u $\leftarrow$ v se e somente se v $\rightarrow$ u;
- $\leftrightarrow = \rightarrow \cup \leftarrow$, de forma que u $\leftrightarrow$ v se e somente se u $\rightarrow$ v ou u $\leftarrow$ v
- $\rightarrow^n$ a relação que é definida indutivamente por
  - u $\rightarrow^0$ v se e somente se u $\equiv$ v e,
  - u $\rightarrow^{n+1}$ v se e somente se $\exists$w, u $\rightarrow^n$ w $\rightarrow$ v;
- ${}^n\leftarrow$ é a mesma relação indutiva extendida para a inversa;
- $\twoheadrightarrow$ o fecho reflexivo transitivo da relação $\rightarrow$;

Confluência em um sistema de reescrita

Um sistema de reescrita é dito confluente se

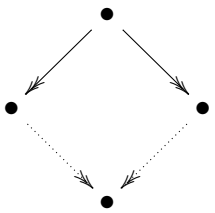
$$(\twoheadleftarrow \circ \twoheadrightarrow) \subseteq (\twoheadrightarrow \circ \twoheadleftarrow).$$

onde $\circ$ representa a composição de duas relações.

Isto significa que para todo u, v, w $\in$ M com v $\twoheadleftarrow$ u $\twoheadrightarrow$ w, existe algum r $\in$ M com v $\twoheadrightarrow$ r $\twoheadleftarrow$ w. O leitor interessado em mais detalhes sobre sistemas de reescrita pode consultar [22].

3.3.1 Confluência no cálculo lambda

Para provar a confluência do cálculo lambda é necessário mostrar que qualquer divergência converge como mostra o esquema abaixo.



No caso de λ_x as setas do esquema anterior podem ser entendidas como o fecho transitivo reflexivo de λ_x. A confluência também é geralmente chamada de propriedade de Church-

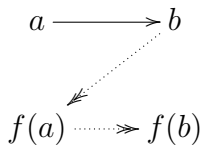
Rosser. Para concluir a prova foi usada a estratégia de prova usando uma extensão do teorema Z, definido em [2], para funções composicionais como formalizado em [7].

3.3.2 A propriedade Z composicional

A propriedade Z é definida como a seguir:

Definição 1 [23] Dadas uma relação binária $\rightarrow$ sobre um conjunto A e uma função $f : A \rightarrow A$, dizemos que f satisfaz a propriedade Z, se $a \rightarrow b$ implica $b \twoheadrightarrow f(a) \twoheadrightarrow f(b)$.

Podemos ver o porquê do nome Z no esquema abaixo:



Em [23], V. van Oostrom e P. Dehornoy mostram que uma relação que possui uma função que satisfaz a propriedade Z é confluente.

Neste trabalho utilizaremos uma versão mais fraca da propriedade Z como definida a seguir:

Definição 2 [2] Dadas duas relações binárias $\rightarrow_1$ e $\rightarrow_2$ sobre um conjunto A e uma função $f : A \rightarrow A$, dizemos que f satisfaz a propriedade Z fraca para $\rightarrow_1$ por $\rightarrow_2$, se $a \rightarrow_1 b$ implica $b \twoheadrightarrow_2 f(a) \twoheadrightarrow_2 f(b)$.

Se $\rightarrow$ possui uma função que satizfaz a propriedade Z fraca então ela também é dita monotônica, ou seja, se $a \rightarrow b$ então $f(a) \rightarrow f(b)$ o que pode ser provado por indução.

A propriedade Z composicional é dada de acordo com o seguinte teorema:

Teorema 1 ([2])**.** *Sejam $(A, \rightarrow)$ um sistema de reescrita abstrato, e $\rightarrow = \rightarrow_1 \cup \rightarrow_2$. Se existem as funções $f_1, f_2 : A \rightarrow A$ tais que*

(a) f_1 satisfaz Z para $\rightarrow_1$

(b) $a \rightarrow_1 b$ implica $f_2(a) \twoheadrightarrow f_2(b)$

(c) $a \twoheadrightarrow f_2(a)$ vale para qualquer $a \in Im(f_1)$

(d) $f_2 \circ f_1$ *é fracamente Z para* $\rightarrow_2$ *por* $\rightarrow$ *,*

então $f_2 \circ f_1$ *satisfaz Z para (A,* $\rightarrow$ *), portanto (A,* $\rightarrow$ *) é confluente.*

3.3.3 Prova por Z composicional

Tendo o cálculo λ_x como um sistema de reescrita abstrato é possível provar sua confluência via a propriedade Z composicional. Para isso são definidas as funções P e B, que corresponderão às funções f_1 e f_2 do lema anterior, respectivamente:

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

Alguns lemas relativos à função P foram definidos para auxiliar em provas que serão apresentadas mais à frente no texto.

Lema 33 (aeq_swap_P)**.** *aeq (P (swap x y t)) (swap x y (P t)).*

Esse lema teve a sua prova completa, porém depende do Lema 31 que não foi completamente provado ainda.

Lema 34 (aeq_P)**.** *aeq t1 t2 -> aeq (P t1) (P t2).*

Também teve sua prova completa, mas depende dos Lemas 31 e 32 que não foram completamente provados.

Lema 35 (notin_P)**.** *x 'notin' fv_nom t -> x 'notin' fv_nom (P t).*

Lema 36 (notin_P_2)**.** *x 'notin' fv_nom (P t) -> x 'notin' fv_nom t.*

Esses dois lemas pendem apenas no caso da substituição explícita. Como a função P transforma uma substituição explícita em uma metasubstituição, temos que trabalhar com o conjunto das variáveis livres da metasubstituição, e não está muito claro como a prova deve seguir.

Lemas auxiliares

Em [2] foram definidos os seguintes lemas auxiliares para completar a prova:

- (1) $M \rightarrow_\pi N$ implica $M^P = N^P$.
- (2) M^P é puro.

- (3) Se M é puro, então $M^P = M$.
- (4) Se M é puro, então $M[x/N] \twoheadrightarrow_\pi M\{x/N\}$.

Para a representação nominal no ambiente de prova, o item (1) teve que passar por uma adaptação porque a igualdade não é sintática, mas uma igualdade módulo α-equivalência. De fato, quando a redução consiste na regra que propaga a substituição explícita para dentro de uma abstração, precisamos propagar a metasubstituição na abstração para fecharmos o diagrama abaixo, e esta propagação só vale, em geral, módulo α-equivalência.

$$\begin{array}{ccc}
(\lambda y.M)[x/N] & \to_\pi & \lambda y.M[x/N] \\
\Big| P & & \Big| P \\
((\lambda y.M)[x/N])^P & & (\lambda y.M[x/N])^P \\
\Big| =_{def.P} & & \Big| = \\
(\lambda y.M)^P\{x/N^P\} & & \lambda y.(M[x/N])^P \\
\Big| = & & \Big| = \\
(\lambda y.M^P)\{x/N^P\} & \equiv_\alpha & \lambda y.M^P\{x/N^P\}
\end{array}$$

No ambiente de prova, o Lema (1) foi traduzido da seguinte forma:

Lema 37 (pi_P)**.** *(ctx pix) t1 t2 -> aeq (P t1) (P t2).*

Em [2] esse lema foi provado informalmente por indução em refltrans (ctx pix) t1 t2. A prova desse lema não foi completada. Aqui o desafio é provar um passo onde existe um *swap* que incide no termo a ser substituído em uma metasubstituição. Outros pontos foram concluídos porém usando o Lema 31 que não foi completamente provado.

Lema 38 (pure_P)**.** *pure (P e).*

Lema 39 (pure_P_id)**.** *pure e -> P e = e.*

Esses dois lemas são triviais e tiveram a sua prova completa por indução em e.

Lema 40 (pure_pix)**.** *pure e1 -> refltrans (ctx pix) (n_sub e1 x e2) (m_subst e2 x e1).*

O lema pure_pix seguiu a estratégia da indução em $e1$ e foi provado em parte faltando apenas um ponto no caso da abstração. Mais uma vez esse lema apresenta um caso onde é necessário manipular a metasubstituição. Abaixo segue o único caso que falta para a prova do Lema 40 ser concluída.

```
- x : atom
- e1 : n_sexp
- x0 : atom
- e2 : n_sexp
- IHe1 : pure e1 -> refltrans (ctx pix) (n_sub e1 - - x0 e2)
      (m_subst e2 x0 e1)
- H1 : refltrans (ctx pix) (n_sub e1 x0 e2)
      (subst_rec (size e1) e1 e2 x0)
- n : x <> x0
- H2 : pure e1
- x1 : atom
- n0 : x1 `notin` Metatheory.union (fv_nom e2)
      (Metatheory.union (remove x (fv_nom e1))
      (singleton x0))
      n1 : x0 <> x
      n2 : x <> x1
============================================================
   aeq (n_abs x (subst_rec (size e1) e1 e2 x0))
      (n_abs x1 (subst_rec (size e1) (swap x x1 e1) e2 x0))
```

Por fim a estrutura da prova da confluência é concluída com o Lema lambda_x_Z_comp_eq definido em ZtoConfl.v.

```
exists (R1 R2 : Rel n_sexp) (f1 f2 : n_sexp -> n_sexp),

    (forall x y : n_sexp, lx x y <-> (R1 !_! R2) x y) /\

    (forall a b : n_sexp, R1 a b -> f1 a = f1 b) /\

    (forall a : n_sexp, refltrans R1 a (f1 a)) /\

    (forall b a : n_sexp, a = f1 b -> refltrans lx a (f2 a)) /\

    f_is_weak_Z R2 lx (f2 # f1).
```

onde R1 e R2 serão aplicadas com regras pix e betax, e as funções f1 e f2 serão aplicadas com as funções P e B.

Capítulo 4

Conclusão

Neste documento estudamos uma abordagem nominal para uma extensão do cálculo lambda com substituições explícitas, o porquê do advento das substituições explícitas, como é feita a representação nominal e passamos por alguns lemas e propriedades desta abordagem. Vimos também a estrutura da prova da confluência do cálculo λ_x pela propriedade Z composicional baseada no trabalho de Nakazawa e Fujita [2] e passamos por alguns lemas auxiliares definidos para concluir a prova partindo do trabalho de Moura e Rezende em [7].

O objetivo desse trabalho era a adaptação da estrutura da representação nominal do cálculo lambda no trabalho de [8] para incluir substituições explícitas na definição do cálculo e após isso a conclusão da prova da confluência. No decorrer da prova porém, foram encontrados alguns casos que poderiam ser simplificados com a prova de alguns lemas auxiliares, porém muitos deles também não eram triviais e demandavam uma infraestrutura de prova mais robusta. Por fim, o objetivo desse trabalho foi adaptado para entregar um conjunto de lemas e propriedades úteis para a conclusão da prova da confluência

Apesar de próximo da representação do papel e lápis, a representação nominal apresenta alguns empecilhos para o ambiente formal de prova. Um deles é a necessidade de ter que lidar explicitamente com a relação de α-equivalência, para isso foi definida a operação de *swap* e um dos desafios desse trabalho foi manipular provas que envolviam uma manipulação de termos com *swap*. Foi necessário definir muitos lemas auxiliares para concluir provas de lemas como `aeq_swap` e `aeq_sym`.

Como trabalho futuro, visando obter a formalização completa da confluência do cálculo λ_x no contexto nominal, temos os seguintes pontos a serem abordados:

- Prova do Lema pi_P: Acreditamos que pode ser concluída por indução em $M \to_\pi N$, mas é preciso primeiramente provar as suas dependências aeq_subst_same, aeq_P e aeq_swap_P.
- Prova do Lema pure_pix: Acreditamos que esteja perto de ser concluída e talvez possa ser provada definindo algum lema auxiliar que associe uma metasubstituição com um *swap* no termo a sofrer substituição.

Em resumo os pontos principais que faltam serem provados dependem de uma manipulação na metasubstituição que não está muito clara.

O repositório desse trabalho pode ser encontrado em `https://github.com/flaviodemoura/lx_confl/`.

Referências

[1] Church, Alonzo: *A set of postulates for the foundation of logic*. Annals of Mathematics, 33(2):346–366, 1932. https://doi.org/10.2307/1968337. 1, 3

[2] Nakazawa, Koji e Ken etsu Fujita: *Compositional z: Confluence proofs for permutative conversion*. 2016. 1, 22, 25, 26, 27, 29

[3] Lins, R. D.: *A new formula for the execution of categorical combinators*. Em Siekmann, Jörg H. (editor): *8th International Conference on Automated Deduction*, páginas 89–98, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg, ISBN 978-3-540-39861-5. 1, 6

[4] Lins, Raphael D: *Partial categorical multi-combinators and church-rosser theorems*. Relatório Técnico 7-92*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, May 1992. http://www.cs.kent.ac.uk/pubs/1992/105. 1, 6

[5] Rose, Kristoffer Høgsbro: *Explicit cyclic substitutions*. Em Rusinowitch, M. e J. L. Rémy (editores): *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 de *LNCS*, páginas 36–50, Pont-a-Mousson, France, julho 1992. Springer. 1, 6

[6] Bloo, Roel e Kristoffer H. Rose: *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection*. Em *CSN '95—Computing Science in the Netherlands*, páginas 62–72, Koninklijke Jaarbeurs, Utrecht, novembro 1995. 1, 6

[7] Moura, Flávio L. C. de e Leandro O. Rezende: *A formalization of the (compositional) z property*. Em *Fifth workshop on formal mathematics for mathematicians*, 2021. http://flaviomoura.info/files/reportZtoConfl.pdf. 1, 25, 29

[8] *Lecture material for deepspec summer school 2017*. https://github.com/DeepSpec/dsss17. Accessed: 2021-10-04. 1, 2, 13, 29

[9] Kleene, Stephen e John Barkley Rosser: *The inconsistency of certain formal logics*. Mathematische Annalen, 36(3):630 – 636, 1935. https://www.jstor.org/stable/1968646. 3

[10] Church, Alonzo e J. B. Rosser: *Some properties of conversion*. Transactions of American Mathematical Society, páginas 472 – 482, 1936. https://www.ams.org/journals/tran/1936-039-03/S0002-9947-1936-1501858-0/. 3

[11] Moura, Flávio Leonardo Cavalcanti de: *Comparando cálculos de substituições explícitas com eta-conversão*, 2002. http://flaviomoura.info/files/Mou02.pdf. 4

[12] Abadi, Martín, Luca Cardelli, Pierre Louis Curien e Jean Jacques Levy: *Explicit substitutions*. Journal of Functional Programming, 1:375 – 416, outubro 1991. 6

[13] Kesner, Delia: *The theory of calculi with explicit substitutions revisited*. Em Duparc, Jacques e Thomas A. Henzinger (editores): *Computer Science Logic*, páginas 238–252, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg, ISBN 978-3-540-74915-8. 6

[14] Aydemir, Brian, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack e Stephanie Weirich: *Engineering formal metatheory*. 2008. 6, 8, 13

[15] Charguéraud, Arthur: *The locally nameless representation*. Journal of Automated Reasoning, 2011. https://charglueraud.org/research/2009/ln/main.pdf. 7

[16] Bruijn, N. G. de: *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem*. Indagationes Mathematicae (Proceedings), 1972. https://doi.org/10.1016/1385-7258(72)90034-0. 7

[17] 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, ISSN 1571-0661. https://www.sciencedirect.com/science/article/pii/S1571066107002319, Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006). 7

[18] *Coq manual of reference*. https://coq.inria.fr/distrib/current/refman/, Acessed: 2021-10-15. 9, 10

[19] Pierce, Benjamin C., Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg e Brent Yorgey: *Software Foundations*, volume 1. https://softwarefoundations.cis.upenn.edu/lf-current/index.html. 9

[20] Bertot, Yves: *Coq in a Hurry*. https://cel.archives-ouvertes.fr/inria-00001173, Lecture, maio 2010. 12

[21] https://github.com/plclub/metalib. 13

[22] Terese: *Term Rewriting Systems*, volume 55 de *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003. 24

[23] 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. 25