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

Em direção à formalização das propriedades de normalização do sistema λex

Lucas de Moura Amaral

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

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

Brasília
2016

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

Em direção à formalização das propriedades de normalização do sistema λex

Lucas de Moura Amaral

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

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

Prof. Dr. Mauricio Ayala-Ríncon
MAT/CIC/UnB

Prof. Dr. Daniel Lima Ventura
INF/UFG

Prof. Dr. Rodrigo Bonifácio de Almeida
Coordenador do Bacharelado em Ciência da Computação

Brasília, 09 de Dezembro de 2016

Dedicatória

Dedico este trabalho à memória do meu pai, Gesley, que sempre fez de tudo para me dar uma boa educação e me incentivou a escolher o curso de Ciência de Computação. Também à minha família, que sempre me ajuda em tudo que preciso e não deixa de demonstrar interesse em minha vida: à minha mãe, Esther; meu padastro, Márcio; e a meus irmãos Thiago, Davi e Benjamin. Agradeço em especial à minha mãe, meu tio Donney e minha avó Nicinha, por nunca deixar faltar nada para que eu pudesse me concentrar em minha formação.

Dedico também aos meus amigos, tanto da escola quanto da faculdade. Em especial, agradeço ao Calil, à Dani e à minha prima, Tephinha, por aguentarem minhas reclamações e me tranquilizarem nos momentos de ansiedade e cansaço.

Agradecimentos

Agradeço ao Departamento de Ciência da Computação e aos meus professores pela boa base técnica dada. Em especial, agradeço ao meu orientador, Flávio, pela disponibilidade e paciência para me ajudar no trabalho.

Resumo

O cálculo λ é um sistema formal, capaz de expressar o processo computacional. Pela sua simplicidade e expressividade, este cálculo é usado como modelo teórico para o paradigma de programação funcional. Em consequência disto, uma grande quantidade de extensões do cálculo foi proposta, com o objetivo de obter um sistema formal intermediário entre o cálculo λ e suas implementações. O objeto de estudo deste trabalho é uma destas variantes, chamada λex, um cálculo com substituições explicitas proposto por Delia Kesner. Este cálculo é um dos primeiros a possuir a preservação da normalização forte enquanto permite composição completa de substituições explícitas. Continuamos o trabalho de formalização deste cálculo, no assistente de prova Coq, iniciado em 2014, e que tem por objetivo fornecer uma prova mecância e construtiva da propriedade de normalização forte para o cálculo λex. Mais especificamente, iniciamos a prova da propriedade IE, chave para a prova da preservação da normalização forte do cálculo λex. Isto foi feito seguindo a estratégia de prova no artigo da Kesner: estendemos a formalização para marcar alguns termos que não inserem problemas de normalização e definimos regras de redução para lidar com tais termos. Por fim, provamos a equivalência dessas novas regras com a regra original do sistema.

Palavras-chave: cálculo lambda, verificação formal, substituições explícitas, Coq

Abstract

λ-calculus is a formal system, capable of expressing the computational process. Because of its simplicity and expressiveness, this calculus is used as a theorical model for the paradigm of functional programming. Consequently, a great variety of extensions were proposed, with the goal of obtaining an intermediate formal system between the λ-calculus and its implementations. The object of study of this work is one of these variants, called λex, a calculus with explicit subsititutions, proposed by Delia Kesner. This calculus is one of the first to preserve strong normalization of terms while permitting full composition of explicit substitutions. We continued the work in the formalization of this calculus, in the Coq proof assistant, initiated in 2014, with the goal of providing a mechanical and constructive proof of the strong normalization property for the λ ex calculus. More specifically, we began the proof of the IE property, key to the demonstration of the preservation of strong normalization of the λex-calculus. This was done following the strategy on Kesner's paper: we extended the formalization to mark some terms that do not insert normalization issues and define reduction rules to deal with such terms. Finally, we prove the equivalence of these new rules with the original reduction rule of the system.

Keywords: lambda calculus, formal verification, explicit substitutions, Coq

Sumário

Capítulo 1

Introdução

1.1 Assistentes de Prova

1.1.1 Motivação

Assistentes de provas são sistemas computacionais que permitem aos usuários especificar teorias e provar propriedades destas em um computador. Neles, o usuário pode construir toda sua teoria matemática em uma linguagem em que o sistema seja capaz de verificar sua correção. Ou seja, a principal motivação por trás de um assistente de prova é verificar formalmente as propriedades de uma teoria. Apesar de já existir um processo humano de verificação, muitas vezes ocorrem erros neste processo, e provas que foram aceitas numa primeira avaliação são descobertas problemáticas algum tempo depois. Como exemplo, podemos citar o teorema das quatro cores (1), que desafiou matemáticos por anos. Este teorema foi provado primeiramente em 1976, por Appel e Haken. Porém esta foi uma prova incompleta e difícil de ser verificada manualmente. Uma prova do teorema foi formalizada em 2008, no assistente de prova Coq, encerrando debates sobre a aceitação do teorema (2; 3)

Mas, então, o que exatamente significa uma prova? Uma prova normalmente é definida como o processo de se estabelecer a validade de alguma afirmação. Na matemática, costumama-se exigir uma clareza e rigor mais extremo, de modo a reduzir ambiguidades e falhas. Porém, matemáticos são humanos e, infelizmente, cometem erros. Com isto em mente, foi definida uma noção ainda mais forte de prova, chamada *prova formal*.

Uma prova formal é uma árvore finita cujos nós são marcados com fórmulas lógicas. As folhas desta árvore são marcadas com axiomas ou premissas, enquanto que os nós filhos são marcados com fórmulas obtidas a partir dos nós pais via regras de inferência. A vantagem do rigor das provas formais é que conferí-las se torna um processo muito mais simples, sendo necessário apenas confirmar de onde vem cada uma das fórmulas.

Por este motivo, provas formais são comumente construídas e verificadas pelos assistentes de prova. Ao utilizar a linguagem do assistente, ele aos poucos constrói a sequência de sentenças e simultaneamente checa sua validade. Porém, isto gera outra dúvida: Por que confiar nos assistentes de prova?

Lógica do assistente: Os assistentes de prova em geral possuem uma teoria forte na qual são baseados. Em geral, existe um sistema matemático independente de implementação que pode ser estudado e verificado anteriormente.

Checagem do assistente: O assistente em si é, também, um programa. Assim, podemos analizar seus procedimentos, demonstrar que só é possível provar teoremas derivados no sistema lógico interno e testar seu funcionamento como um programa normal.

Critério de De Bruijn: O critério de De Bruijn afirma que a correção de um sistema deve ser garantida por um verificador *pequeno* (4). Em outras palavras, deve haver um *kernel* pelo qual todas as fórmulas passam. Assim, se torna mais fácil estabelecer a confiabilidade do sistema, pois podemos verificar o kernel separadamente. Segundo Wiedijk (4), Nem todos os assistentes de prova passam neste critério, como visto na Tabela 1.1.

Tabela 1.1: Relação entre assistentes de prova e o critério de De Bruijn (4)

	Critério de De Bruijn
HOL	•
Mizar	
PVS	
Coq	•
Otter/Ivy	•
Isabelle/Isar	•
Alfa/Agda	•
ACL2	
PhoX	•
IMPS	
Metamath	•
Theorema	
Lego	•
NuPRL	
Ωmega	•

Há um crescente interesse em assistentes de prova, onde busca-se construir uma teoria consistente para o uso destes e se produzo software necessário para facilitar seu uso. Em especial, um dos assistentes com maior uso é o chamado Coq, a ser apresentado a seguir.

Para uma visão geral sobre o histórico e uso de assistentes de prova, veja (5).

1.1.2 O assistente de prova Coq

Neste trabalho, usaremos o Coq, um assistente de provas que é desenvolvido desde 1983, pelo INRIA (Institut-National de Recherche en Informatique et en Automatique). O Coq provê um rico ambiente para o desenvolvimento de um raciocínio formal checado automaticamente. O núcleo do sistema é um checador de provas simples que garante que apenas passos válidos de dedução são efetuados. Além deste núcleo, o ambiente provê diversas táticas para facilitar a construções de provas, junto com uma vasta biblioteca de definições e teoremas comuns.

A ferramenta vem acompanhada de uma linguagem de especificação funcional, com tipos dependentes. É através desta linguagem que podemos criar as definições e construir as provas das propriedades de nossa teoria. Ela é baseada no Cálculo de Construções Indutivas (6), uma extensão do cálculo λ que serve como modelo teórico para o sistema. O processo de se verificar a correção de uma prova em Coq se reduz ao problema de *checagem de tipos*. A seguir, será feita uma introdução à sintaxe e ao funcionamento da ferramenta, baseada em tutoriais disponibilizados na página oficial do sistema, em (7) e (8).

Os objetos de Coq podem ser divididos em duas categorias, *Prop* e *Type*. A categoria *Prop* é a das proposições bem formadas. Um exemplo de proposição na linguagem seria:

$$\forall\ A\ B : \texttt{Prop},\ A \to (A \vee B).$$

Predicados podem ser definidos indutivamente. Na definição a seguir, `even` é um predicado que indica que um natural é par, e `odd` indica que um natural é ímpar.

$$\begin{aligned}
&\texttt{Inductive}\ even : nat \to \texttt{Prop} :=\\
&\ \mid even_0 : even\ 0\\
&\ \mid even_S\ n : odd\ n \to even\ (n+1)\\
&\texttt{with}\ odd : nat \to \texttt{Prop} :=\\
&\ \mid odd_S\ n : even\ n \to odd\ (n+1).
\end{aligned}$$

Predicados também podem ser especificados como definições diretas, como:

$$\texttt{Definition}\ sqr\ (x : N) := \exists\ z,\ z \times z = x.$$

Assim, podemos utilizar estes predicados na identificação de propriedades de algum objeto, provando algo como *even(2)* ou *sqr(4)*.

Type é a categoria de estruturas matemáticas e estruturas de dados. Alguns exemplos de tipos são:

$$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

Este é um tipo funcional. $\mathbb{Z} \times \mathbb{Z}$ é o tipo de pares de números inteiros. O tipo completo, $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, consiste de funções de pares de inteiros que retornam inteiros.

Tipos também podem ser definidos indutivamente:

```
Inductive nat : Set :=
 | 0 : nat
 | S : nat → nat.
```

Neste caso, elementos do tipo *nat* são: 0, S (0), S (S (0)), etc.

O desenvolvimento de provas em Coq é feito através de uma linguagem de provas, que permite um processo guiado pelo usuário. Ao utilizar uma tática, o usuário está construindo os objetos de prova. Por exemplo, a tática `intro n`, onde `n` é do tipo `nat`, constrói o termo:

```
fun (n:nat) ⇒ _
```

Onde _ representa um termo que irá ser construído futuramente, utilizando outras táticas.

Neste trabalho, precisaremos também definir algumas funções recursivas. Para garantir a consistência do sistema, a ferramenta exige que todas as funções sejam terminantes. Um exemplo da sintaxe usada para definir funções no trabalho pode ser visto abaixo.

```
Fixpoint mult_5 (n : nat) : nat :=
    match n with
    | S k ⇒ 5 + (mult_5 k)
    | 0 ⇒ 0
    end.
```

A função é chamada *mult_5* e recebe um natural. Ela realiza uma análise de casos no argumento, observando seus construtores. Caso o número seja zero, a função retorna zero. Se for o sucessor de algum natural k, ela soma 5 ao resultado de *mult_5* k. O sistema consegue deduzir a terminação desta função, pois a chamada recursiva é realizada em um subtermo do argumento inicial.

Como exemplo de uma prova simples a ser realizada no sistema, podemos comparar o funcionamento da nossa função com a multiplicação padrão do Coq.

```
Theorem mult_5_is_correct: ∀ n : nat, (mult_5 n) = 5*n.
Proof.
    intros.
    induction n.
    simpl. reflexivity.
    simpl mult_5.
    rewrite mult_comm.
    simpl.
    rewrite IHn.
    rewrite mult_comm.
    reflexivity.
Qed.
```

No exemplo acima, `mult_5_is_correct` é o nome do teorema, e `Proof/Qed` delimita a prova. A ferramenta disponibiliza uma maneira de visualizar os estados da prova durante o processo. Introduzimos um natural arbitrário para começar a prova, com o comando `intros n`. Após isto, realizamos uma prova por indução com o comando `induction n`, que divide a prova em dois subcasos. O primeiro corresponde à base de indução, e é dado por:

```
============================ ( 1 / 2 )
mult_5 0 = 5 * 0
```

Usamos o comando `simpl` para computar os valores de `mult_5 0` e `5 * 0`, tendo assim a igualdade `0 = 0`. A validade disso vem da reflexividade da igualdade, concluindo este caso com o comando `reflexivity`. No passo indutivo, temos o seguinte estado:

```
n : nat
IHn : mult_5 n = 5 * n

============================ ( 1 / 1 )
mult_5 (S n) = 5 * S n
```

Onde `IHn` é nossa hipótese de indução. Começamos a prova computando o valor de `mult_5 (S n)`, com o comando `simpl mult_5`. Observando a definição desta função, vemos que o a expressão será avaliada para `S (S (S (S (mult_5 n))))`, refletido no estado da prova. Queremos então avaliar também o resultado de `5 * S n`. Devido à definição do operador *, a prova se torna mais fácil se comutarmos os termos `5` e `S n` nesta expressão. Fazemos isto utilizando o lema `mult_comm`, pertencente às bibliotecas do sistema, com o comando `rewrite mult_comm`. Após isto, computamos o valor de `(S n) * 5`, com o comando `simpl`. Temos, então, o seguinte estado:

```
n : nat
IHn : mult_5 n = 5 * n

============================ ( 1 / 1 )
S (S (S (S (S (mult_5 n))))) = S (S (S (S (S (n * 5)))))
```

Podemos finalmente usar nossa hipótese de indução! Reescrevemos o termo `mult_5 n` para `5 * n`, com o comando `rewrite IHn`. Utilizamos novamente a comutatividade do operador *, tendo finalmente o seguinte estado:

```
n : nat
IHn : mult_5 n = 5 * n

============================ ( 1 / 1 )
S (S (S (S (S (n * 5))))) = S (S (S (S (S (n * 5)))))
```

Agora temos o mesmo termo em ambos os lados da igualdade, podendo concluir a prova com `reflexivity`.

Os conceitos apresentados nesta seção serão amplamente utilizados no trabalho. Para um melhor entendimento do funcionamento da ferramenta, da linguagem e teoria envolvidas, veja (9).

1.2 O cálculo λ

1.2.1 Visão geral

Um dos grandes desafios do início do século XX era obter uma definição precisa para a noção intuitiva do funções computáveis. Diversos modelos foram propostos para resolver este problema. Entre eles está o cálculo λ, proposto por Alonzo Church (10), em 1936. Inicialmente, o cálculo fazia parte de um sistema maior, proposto para servir como uma base formal para o estudo das fundações da matemática. Porém, devido a inconsistências neste sistema, Alonzo Church foi obrigado a abrir mão de seu objetivo inicial, separou a parte utilizável do sistema e formou o que hoje conhecemos como cálculo λ.

O grande diferencial deste cálculo está em sua expressividade, com poder computacional equivalente ao das máquinas de Turing universal, e sua simplicidade, demonstrada por sua gramática concisa e poucas regras. A ideia central deste consiste em simular a criação e aplicação de funções. As funções neste sistema são chamadas "anônimas", pois são definidas tendo em vista somente seus argumentos e o resultado. Como exemplo, uma função simples como: "$double(x) = 2 * x$" é definida anonimamente como "$\lambda x.\ 2 * x$".

É utilizada também uma notação especial para a aplicação de funções, denotada como "$(\lambda x.\ 2 * x)\ 3$".

É importante notar que os exemplos acima não são representados exatamente desta maneira. Como dito anteriormente, o sistema possui uma gramática simples, e todas as noções, inclusive números e operações, devem ser definidas com base em abstrações e aplicações. A gramática do cálculo λ pode ser descrita sucintamente como:

$$\tau := x \mid \lambda x.\tau \mid \tau\tau$$

Onde τ representa um termo, e x representa uma variável. Assim, um termo do cálculo pode ser, respectivamente, uma variável, uma abstração de um termo τ por uma variável x, ou uma aplicação de dois termos.

Definição 1.1. *Definimos o conjunto de variáveis livres de t, denotado por $fv(t)$, indutivamente. Na definição abaixo, t e u denotam termos, enquanto x denota uma variável.*

$$\begin{aligned} fv(x) &= \{x\} \\ fv(t\ u) &= fv(t) \cup fv(u) \\ fv(\lambda x.t) &= fv(t) \setminus \{x\} \end{aligned}$$

Dizemos que uma variável x é livre em t se $x \in fv(t)$.

Como exemplo, a variável x é livre em $(\lambda y.\ y)\ x$, enquanto a variável y não é.

O processo computacional é simulado no sistema através da regra de β-redução, definida como:

$$(\lambda x.t)\ u \to_\beta t\{x/u\}$$

onde $t\{x/u\}$ denota uma *meta-operação* de substituição, definida como a substituição das ocorrências livres da variável x no termo t pelo termo u. Esta operação será definida formalmente em breve.

Para facilitar o entendimento, vejamos alguns exemplos de λ-termos:

- A função identidade pode ser representada pelo termo $(\lambda x.x)$. É fácil ver a correspondência na seguinte redução: $(\lambda x.x)u \to_\beta x\{x/u\} = u$.
- A função constante pode ser representada pelo termo $(\lambda x.M)$, onde M é um termo qualquer, tal que $x \notin fv(M)$. É fácil ver a correspondência na seguinte redução: $(\lambda x.M)u \to_\beta M\{x/u\} = M$.
- Por último, podemos representar uma função que recebe dois termos e retorna o primeiro, como $(\lambda x.\lambda y.x)$. Sua aplicação é reduzida da seguinte maneira: $((\lambda x.\lambda y.x)\ M)\ N) \to_\beta$

$((\lambda y.x)\{x/M\}N) = (\lambda y.M)N \rightarrow_\beta M\{y/N\} = N$, com x,y não ocorrendo livres em M ou N.

A partir destas definições, várias propriedades sobre o sistema podem ser estudadas. Entre elas, é importante ressaltar as noções de *forma normal* e de *confluência*:

Definição 1.2 (Forma normal)**.** *Um termo t é dito estar em forma normal quando não existe t' tal que $t \rightarrow_\beta t'$. Um termo é dito* ***fracamente normalizável*** *quando existe uma estratégia de redução que leva a uma forma normal. O termo é* ***fortemente normalizável*** *se toda estratégia leva à forma normal.*

Nem todo termo possui uma forma normal. Como exemplo, observe que $(\lambda x.x\ x)\ (\lambda x.x\ x) \rightarrow_\beta (\lambda x.x\ x)\ (\lambda x.x\ x)$. A noção de normalização é especialmente importante, pois indica se um termo pode ou não terminar quando for avaliado, o que é de grande interesse no estudo de propriedades computacionais. Isto porque a normalização está intimamente relacionada à noção de terminação de programas. Um termo fortemente normalizável é correspondente a um algoritmo, ou seja, sempre termina.

Definição 1.3 (Confluência)**.** *Um sistema de reescrita, tomando como exemplo o cálculo λ, é dito fracamente confluente se, para todo termo t, com $t \rightarrow_\beta t'$ e $t \rightarrow_\beta t''$, então deve existir um termo u tal que $t' \rightarrow_\beta^* u$ e $t'' \rightarrow_\beta^* u$, onde $\rightarrow_\beta^*$ representa o fecho transitivo-reflexivo da β-redução. O sistema é dito fortemente confluente se, para todo termo t tal que $t \rightarrow_\beta^* t'$ e $t \rightarrow_\beta^* t''$, existe um termo u tal que $t' \rightarrow_\beta^* u$ e $t'' \rightarrow_\beta^* u$.*

A propriedade de confluência pode ser entendida, essencialmente, como uma garantia que a ordem que as reduções são feitas não afetam a forma normal do termo. Ou seja, a confluência garante o determinismo do processo computacional em termos fortemente normalizáveis.

Outro ponto importante a ser mencionado é a noção de α-equivalência de termos.

Definição 1.4 (α-equivalência)**.** *Definimos a relação de α-equivalência indutivamente na estrutura dos termos:*

$$
\begin{aligned}
x &=_\alpha x \\
u\ v &=_\alpha s\ t \iff (u =_\alpha s) \wedge (v =_\alpha t) \\
\lambda x.s &=_\alpha \lambda y.t \iff (s\{x/y\} =_\alpha t) \wedge (y \notin fv(s))
\end{aligned}
$$

Esta noção captura a ideia de que a escolha do nome das variáveis ligadas não importa em geral, sendo o real objeto de interesse a estrutura do termo. Esta noção é útil para evitar certos problemas, como por exemplo o de *captura de variáveis livres*. No exemplo:

$$(\lambda x.\ y)\{y/x\},\ x \neq y$$

Efetuando a substituição sem tomar o devido cuidado, teremos o termo $\lambda x.\ x$, que representa a função identidade. Note que isto mudou completamente a semântica do termo! O termo anterior, $\lambda x.\ y$, representava uma função constante, que sempre é avaliada para y, independente do termo dado como argumento. Aplicando a substituição, seria razoável esperar que teríamos uma função que é sempre avaliada para a variável x, o que não é o caso.

Veja que a variável x na substituição não é a mesma da que está ligada na abstração. Isto pode causar vários problemas inesperados, como, por exemplo, a perda da propriedade de normalização do termo. Como exemplo, considere o termo $((\lambda x.\ y\ y)\ (\lambda x.\ y\ y))$. É fácil ver que este termo deveria avaliar para $(y\ y)$. Se queremos substituir y por x, deveríamos obter o termo $(x\ x)$. Porém, aplicando a substituição no termo original, temos $((\lambda x.\ y\ y)\ (\lambda x.\ y\ y))\{y/x\}$, que é avaliado para $((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$, que não possui forma normal!

Para evitar este problema, podemos renomear a variável ligada x, antes da substituição, por uma variável nova, de maneira a obter um termo α-equivalente e podendo então realizar a substituição sem modificar a semântica do termo. Nos exemplos anteriores:

$$(\lambda x.\ y)\{y/x\}\ =_\alpha (\lambda z.\ y)\{y/x\} = (\lambda z.\ x)$$

Observe que chegamos ao resultado esperado, ou seja, a função constante que é sempre avaliada para x.

$$((\lambda x.\ y\ y)\ (\lambda x.\ y\ y))\{y/x\} =_\alpha ((\lambda z.\ y\ y)\ (\lambda z.\ y\ y))\{y/x\} = ((\lambda z.\ x\ x)\ (\lambda z.\ x\ x)) \rightarrow_\beta (x\ x)$$

Conseguimos, também, evitar a perda da normalização do termo.

Assim, podemos definir formalmente a operação de substituição, da seguinte maneira:

Definição 1.5. *Sejam t, u termos e x uma variável. A substituição de x por u em t é definida indutivamente na estrutura de t como abaixo, trabalhando módulo α-equivalência:*

$$\begin{aligned} y\{x/u\} &= u,\ se\ (x = y);\ y,\ caso\ contrário \\ (t'\ v)\{x/u\} &= (t'\{x/u\}\ v\{x/u\}) \\ (\lambda y.t')\{x/u\} &= (\lambda y.t'), se(x = y); (\lambda y.t\{x/u\}),\ caso\ contrário \end{aligned}$$

Note que, no caso da abstração, se $y \in fv(u)$, devemos realizar a aplicar o renomeamento de y por uma variável nova no contexto, para preservar a semântica do termo.

1.2.2 Representação de λ-termos

Como dito anteriormente, o cálculo λ é usado como modelo teórico para linguagens funcionais. A noção de α-equivalência, apesar de útil, pode ser muito custosa em uma implementação prática. Por este motivo, foram propostas algumas representações diferentes de termos, de modo a instrumentalizar a necessidade do renomeamento de variáveis.

Uma das primeiras, e mais importantes, tentativas de resolver o problema da α-conversão é a notação utilizando índices de De Bruijn (11). Nela, são utilizados números naturais para representar as variáveis. Cada número representa a quantidade de abstrações no escopo da ocorrência da variável. Números que ultrapassam esta quantidade representam variáveis livres. A gramática pode ser definida como:

$$\tau := n \mid \lambda\tau \mid \tau\tau$$

Onde n representa um número natural, a partir do 1. Exemplos de termos nesta notação incluem a identidade ($\lambda 1$), a função constante (λu), com u não contendo 1 como índice livre, e um termo com índice livre, como $((\lambda 2)\ u)$. Apesar de sua aparente praticidade de implementação, esta notação se afasta muito da utilização do cálculo no papel. Além disto, introduz a necessidade de se manter um contexto externo para registrar as variáveis livres, junto com uma álgebra para lidar com tal contexto. Para lidar com substituições utilizando a notação de De Bruijn, devemos introduzir a noção de i-elevação.

Definição 1.6 (i-elevação (12))**.** *Seja t um termo da gramática de De Bruijn. A i-elevação de t, denotada por t^{+i}, é definida indutivamente como:*

- $(t_1\ t_2)^{+i} = (t_1^{+i}\ t_2^{+i})$
- $(\lambda t_1)^{+i} = \lambda(t_1^{+(i+1)})$
- $n^{+i} = \begin{cases} n+1 & \text{se } n > i \\ n & \text{se } n \leq i \end{cases}$

A elevação de um termo t é sua 0-elevação, e é denotada por t^+.

A definição de substituição deve ser adaptada para lidar com os índices de De Bruijn.

Definição 1.7 ((12))**.** *A substituição de um termo t, no nível $n-1$, pelo termo u, denotada por $t\{n/u\}$, é definida indutivamente como:*

- $(t_1\ t_2)\{n/u\} = (t_1\{n/u\}\ t_2\{n/u\})$
- $(\lambda t_1)\{n/u\} = \lambda t_1\{n+1/u^+\}$

- $m\{n/u\} = \begin{cases} m-1 & se\ m > n \\ u & se\ m = n \\ m & se\ m < n \end{cases}$

Temos então que a relação de β-redução é definida por $(\lambda t_1)\ t_2 \rightarrow_\beta t_1\{1/t_2\}$.

Como exemplo das complexidades introduzidas por esta notação, podemos observar a representação do termo $(\lambda x.\ x\ a)\ b$, onde a, b são variáveis livres. Este termo é representado na notação de De Bruijn como:

$$(\lambda\ 1\ 2)\ 2$$

Junto com o contexto $[a,\ b]$, que representa as variáveis livres. Utilizando esse contexto, vemos que o índice 1 representa a variável x, pois temos apenas uma abstração. Após isso, cada ocorrência do índice 2 representa uma variável livre distinta que estão relacionadas, em ordem, com as variáveis no contexto. Assim, a primeira ocorrência do índice 2 deve ser substituída por a, e a segunda deve ser substituída por b.

Após realizar a β-redução, o termo original é representado por $(b\ a)$. Porém, o termo na notação da representação de De Bruijn não é representado por (2 2), mas sim por (2 1), devido às manipulações efetuadas pela substituição. Isto é necessário porque o novo termo não possui abstratores, logo o primeiro índice para representar variáveis livres não é 2, mas 1.

Temos então, dois problemas: utilizar a notação padrão do cálculo λ, junto com α-equivalência, é algo custoso do ponto de vista computacional, pois devemos sempre tomar o cuidado com captura de variáveis livres e fazer o renomeamento destas. Por outro lado, utilizar a notação de De Bruijn também exige cuidados, devido à necessidade de se manter um contexto externo para representar variáveis livres e de se utilizar uma álgebra mais complexa ao realizar substituições.

Uma solução para estes problemas é usar a *locally nameless representation* (LNR), usada para representar os λ-termos neste trabalho. Esta representação tenta capturar o melhor de ambas as abordagens: não temos a necessidade de realizar renomeamento constante de variáveis, nem precisamos lidar com um contexto externo.

O conjunto Λ_{LNR} de expressões na notação LNR é definido a partir da gramática a seguir:

$$\tau := x \mid n \mid \lambda\tau \mid \tau\tau$$

Como nos casos anteriores, x representa uma variável; n é um índice, representado por um natural positivo; $\lambda\tau$ representa uma abstração e $\tau\tau$ representa uma aplicação.

Os índices n representam as variáveis ligadas da expressão. O valor do índice representa a abstração a qual ele se refere, ou seja, um índice k está ligado à k-ésima abstração que o contém.

Dizemos que uma expressão é um pré-termo se pertence ao conjunto Λ_{LNR} gerado pela gramática acima. Observe que, como os índices representam as variáveis ligadas, não é interessante a ocorrência de índices soltos, ou seja, sem uma abstração correspondente. Por conta disto, pré-termos que possuírem índices soltos não terão termos correspondentes na representação padrão do cálculo λ. Definimos então a noção de termos, que serão as expressões sem índices livres, foco de interesse no trabalho.

Definição 1.8 (Termos)**.** *Dizemos que um pré-termo é um termo (bem formado) se toda ocorrência de um índice pertence a pelo menos um número correspondente de abstrações. Em outras palavras, um termo não possui índices soltos.*

A meta-substituição deve ser adaptada para levar em conta os índices, como no exemplo:

$$\{1 \to t\}(\lambda\ 2) = (\lambda(\{2 \to t^+\}2))$$

Note que o índice a ser substituído foi incrementado quando a substituição entra na abstração, de modo a corresponder à variável correta a ser substituída. Precisamos, então, definir formalmente a operação.

Definição 1.9. *Sejam t, u, v termos; y uma variável e k, i índices. A operação de substituição é definida indutivamente como:*

$$\begin{aligned}
\{k \to v\}i &= v,\ \textit{se}\ (i = k);\ i,\ \textit{caso contrário} \\
\{k \to v\}y &= y \\
\{k \to v\}(t\ u) &= (\{k \to v\}t\ \{k \to v\}u) \\
\{k \to v\}(\lambda t) &= (\lambda\{k+1 \to v^+\}t)
\end{aligned}$$

Esta representação foi detalhada por Arthur Charguéraud em (13), junto com provas de seu bom funcionamento e um framework para sua utilização em Coq. Entraremos agora em alguns detalhes do uso desta notação, descritos em (13), já que ela tem um interesse especial neste trabalho.

Na notação usual, quando queremos estudar o corpo de uma abstração $(\lambda x.t')$, podemos trabalhar com o termo t'. Porém, nesta nova representação, a abstração tem a forma $(\lambda\ t')$, e é necessário que seja fornecida uma variável x, já que t' não é um termo, em geral. Esta operação é chamada *abrir o termo t' com x* e será representada por t'^x. Mais precisamente, a abertura do termo $(\lambda\ t')$ cria uma cópia de t' onde todas as ocorrências do

índice ligado à abstração mais externa são substituídos pela variável x. Como exemplo, abrir a abstração $(\lambda(1\ y))$ com x nos dá o termo $(x\ y)$. A operação de abrir o termo deve ajustar o índice a ser mudado à medida que entra no termo. Assim, podemos usar a já definida operação de substituição, diretamente com a variável x, para realizar a abertura. Temos, então, $t'^x = t'\{1 \to x\}$.

Similarmente, podemos abstrair todas as ocorrências de uma variável x no termo t', construindo então o termo correspondente a $(\lambda x.t')$. Com a nova notação, é necessário definir uma operação que substitui todas as ocorrências de x pelo índice 0, antes de adicionar a abstração. Esse processo é chamado *fechar o termo*, representado por $^{\backslash x}t'$. Assim, para construir a abstração de maneira equivalente, fazemos $(\lambda^{\backslash x}t')$.

Definição 1.10. *Sejam t um termo, x uma variável e k um índice. Definimos indutivamente a substituição das ocorrências da variável x em t pelo índice k, denotada por $\{k \leftarrow x\}t$, como a seguir:*

$$\begin{aligned}
\{k \leftarrow x\}i &= i \\
\{k \leftarrow x\}y &= k,\ \textit{se}\ (x = y);\ y,\ \textit{caso contrário} \\
\{k \leftarrow x\}(t\ u) &= (\{k \leftarrow x\}t\ \{k \leftarrow x\}u) \\
\{k \leftarrow x\}(\lambda t) &= (\lambda\{k+1 \leftarrow x\}t)
\end{aligned}$$

Podemos, então, definir a operação de fechamento como $^{\backslash x}t = \{0 \leftarrow x\}t$.

Como dito anteriormente, esta representação possui termos que contém *índices livres*, que não possuem correspondentes no sistema original, e não são nosso objeto de interesse. A seguir, chamaremos um termo sem índices livres, ou seja, um termo bem formado, de *localmente fechado*.

Existem duas maneiras de conferir se um termo é localmente fechado. A primeira consiste em percorrer a estrutura do termo, abrindo cada abstração com uma nova variável. Desta maneira, se o termo for de fato fechado, nunca encontraremos um índice. A segunda abordagem consiste em analizar diretamente os índices do termo, checando, para cada um deles, se o seu valor é menor ou igual ao número de abstrações que o contêm.

A primeira opção induz uma definição natural de um predicado, denotado por $lc\ t$ indicando que o termo é localmente fechado. Com apenas três regras de inferência, podemos facilmente fazer uma análise mais formal da propriedade de ser localmente fechado, sendo bastante útil em provas.

Definição 1.11 (Localmente fechado)**.** *Sejam $t, t1, t2$ termos, x uma variável e L um conjunto finito de variáveis. Dizemos que um termo é localmente fechado se não possui índices livres. Formalmente, isto é feito através do predicado lc, definido pelas regras de inferência abaixo.*

$$\frac{}{lc(x)}\ \text{LC\_FVAR} \qquad \frac{lc\ t1 \qquad lc\ t2}{lc(t1\ t2)}\ \text{LC\_APP} \qquad \frac{\forall x \notin L,\ lc\ (t^x)}{lc(\lambda.t)}\ \text{LC\_ABS}$$

A premissa $\forall x \notin L$ no caso da abstração captura a ideia de x ser uma variável nova, pois podemos tomar o conjunto L, sempre finito, como sendo o conjunto de variáveis já usadas e, assim, sempre ter uma escolha nova de x.

A segunda abordagem tem um caráter mais naturalmente computacional, dando lugar a uma função recursiva para conferir se um termo é localmente fechado:

Definição 1.12. *O predicado binário lc_at que recebe um índice k e um pré-termo t como argumentos é definido indutivamente na estrutura de t, como segue:*

$$\begin{aligned} lc\_at\ k\ (i) &= i < k \\ lc\_at\ k\ (x) &= \textbf{\textit{True}} \\ lc\_at\ k\ (t1\ t2) &= (lc\_at\ k\ t1)\ \&\ (lc\_at\ k\ t2) \\ lc\_at\ k\ (\lambda t1) &= (lc\_at\ k+1\ t1) \end{aligned}$$

Esta função vai navegando pelo termo, entrando em seus subtermos, guardando um contador. Quando entrar em uma abstração, a função incrementa tal contador. Ao encontrar um índice, basta conferir se ele é menor que o contador.

Dizemos que o termo t é localmente fechado se a função $lc\_at\ 0\ t$ retorna **True**. Não é difícil provar a equivalência de ambas as definições, ou seja, que vale $lc\ t \iff (lc\_at\ 0\ t\ =\ \textbf{True})$. Para isto, precisaremos do seguinte lema:

Lema 1.1. *Sejam t um pré-termo e x uma variável. Então vale* $(lc\_at\ k\ \{k \to x\}t) \iff (lc\_at\ (S\ k)\ t)$*, para todo* $k \in \mathbb{N}$.

Teorema 1. *Seja t um pré-termo. Vale (lc t)* $\iff$ *(lc_at 0 t =* ***True****).*

Demonstração.

($\Rightarrow$) Indução no predicado *lc*. O único subcaso que não sai imediatamente é o da abstração, que pode ser facilmente resolvido se observarmos que um termo $\{k \to x\}t$ é fechado a um nível k se, e somente se, t é fechado a nível k + 1, como visto no Lema 1.1.

($\Leftarrow$) Indução na estrutura do termo t. Novamente, o caso da abstração merece um cuidado especial, mas ainda sai de maneira simples, escolhendo o conjunto L referenciado no construtor de *lc* como sendo exatamente o conjunto de variáveis livres

de t. Temos então, como hipótese, que vale $lc\_at\ 1\ t'$, onde t' é o corpo da abstração, e queremos provar $(lc\ t'^x)$, para qualquer $x \notin fv(t')$. Basta então usar novamente o Lema 1.1 para concluir a prova.

□

Estas definições serão extremamente importantes no decorrer do trabalho e podem ser citadas com frequência. Em especial, a definição de pré-termo localmente fechado, correspondente à noção de termo bem formado, é essencial, pois são estes as expressões que possuem termos correspondentes no cálculo λ.

Vale a pena ressaltar duas outras equivalências, referentes às noções de abertura e fechamento de termos.

Lema 1.2. *Seja t um termo e x uma variável. Então $^{\backslash x}(t^x) = t$, se $x \notin fv(t)$*

Lema 1.3. *Seja t um termo e x uma variável. Então $(^{\backslash x}t)^x = t$, se vale $(lc\ t)$*

Para mais informações a respeito do cálculo λ, veja (14).

1.3 Substituições explícitas

1.3.1 Motivação e histórico

Sabemos que a ordem em que as reduções são feitas não altera a forma normal do termo, mas isto não significa que não existem vantagens em se adotar certas estratégias na normalização.

Na implementação de linguagens de programação funcionais, a substituição muitas vezes é "atrasada", de modo a evitar computações desnecessárias. Para aproximar o modelo teórico de seu correspondente prático, podemos fracionar a operação de substituição em partes mais simples, de maneira a permitir uma manipulação simbólica mais precisa. (15)

Além disto, separar a operação de substituição em partes mais simples pode auxiliar no estudo de propriedades do próprio cálculo λ. Uma estratégia comum (16; 17) no estudo de propriedades do cálculo, referentes às substituições, é analisar o problema em uma versão estendida do cálculo λ, que possui um formalismo sintático representando as substituições, permitindo um controle mais preciso destas. Demonstrada a propriedade no novo sistema estendido, basta mostrar que ela é preservada entre os cálculos.

Por este motivo, existem várias tentativas de se formalizar a noção de substituição, dando então espaço para o formalismo conhecido como *substituição explícita*. A princípio, podemos estender a gramática de termos da seguinte maneira:

$$\tau := x \mid \lambda x.\tau \mid \tau\tau \mid \tau[x/\tau]$$

Podemos então espelhar o funcionamento da meta-substituição através de regras de redução no novo cálculo, gerando o sistema conhecido como λx. (18–20)

$(\lambda x.\ t)\ u$	$\rightarrow$	$t[x/u]$
$x[x/u]$	$\rightarrow$	u
$y[x/u]$	$\rightarrow$	$y,\ se\ x \neq y$
$(t\ u)[x/v]$	$\rightarrow$	$t[x/v]\ u[x/v]$
$(\lambda y.\ u)[x/v]$	$\rightarrow$	$(\lambda y.\ u[x/v])$

Tabela 1.2: Regras do sistema λx

O sistema λx corresponde ao comportamento mínimo que se espera de um cálculo com substituições explícitas. Porém, existem outras propriedades interessantes que podem ser adicionadas ao sistema e, em consequência disto, vários outros modelos foram propostos, como o λ_σ (21), λ_{ws} (22), λlxr (23), entre outros.

Um problema que pode acontecer em sistemas com substituição explícita é a perda da preservação da normalização forte (**PSN**). (24; 25)

Definição 1.13 (PSN)**.** *Seja λz uma extensão do cálculo λ. Dizemos que λz preserva a normalização forte se, para todo λ-termo fortemente normalizável, seu correspondente em λz também é fortemente normalizável nesta extensão.*

Este tipo de problema é especialmente comum em cálculos com substituições explícitas que possuem a propriedade de composição de substituições. Essencialmente, dado um termo $t[x/u][y/v]$, podemos comutar as duas substituições, de maneira a reduzir a segunda substituição antes da primeira. Como resultado, teríamos o termo $t[y/v][x/u[y/v]]$.

Várias estratégias são usadas para se tentar garantir a propriedade **PSN** do sistema, como utilização de marcas em termos, restrição de composições ou reduções dentro de substituições explícitas, definições de classes de equivalências, entre outros. Para uma visão geral do histórico de cálculos de substituições explícitas, veja (26).

A seguir, veremos um sistema que propõe uma maneira de se compor tais substituições sem perder a propriedade **PSN**, o chamado λex. O foco deste trabalho será iniciar a formalização da propriedade **IE**, chave para a prova da propriedade **PSN** deste sistema, necessário para completar o trabalho iniciado em (27). Para isto, vamos seguir a estratégia apresentada em (26): *marcaremos* algumas substituições explícitas e iremos definir duas outras reduções, $\rightarrow_{\lambda \underline{ex}^i}$ e $\rightarrow_{\lambda \underline{ex}^e}$, de modo a controlar melhor as interações de termos com substituições explícitas. Por fim, mostraremos a equivalência destas novas reduções com a regra principal do sistema λex.

Capítulo 2

O sistema λex

2.1 Visão geral

Como visto no capítulo anterior, várias extensões do cálculo λ foram propostas com o objetivo de obter um sistema fiel às propriedades deste e onde a operação de substituição fosse um elemento primitivo da linguagem.

O sistema proposto em (26), chamado λex, é o primeiro sistema que captura de maneira simples tal noção, enquanto ainda possui a propriedade **PSN**, ou seja, a preservação da normalização forte. São introduzidas várias mudanças em relação ao cálculo λ, a começar pela gramática:

$$\tau := x \mid \lambda x.\tau \mid \tau\tau \mid \tau[x/\tau]$$

A nova construção é chamada *substituição explícita*, e é o que permite as manipulações sintáticas com substituições no cálculo. Precisamos então estender a definição do conjunto de variáveis livres do novo cálculo.

Definição 2.1. *Na definição abaixo, t, u denotam termos e x denota uma variável. Definimos o conjunto de variáveis livres de t, denotado por $fv(t)$, indutivamente.*

$$\begin{aligned} fv(x) &= \{x\} \\ fv(t'\ u) &= fv(t') \cup fv(u) \\ fv(\lambda x.t') &= fv(t') \setminus \{x\} \\ fv(t'[x/u]) &= (fv(t') \setminus \{x\}) \cup fv(u) \end{aligned}$$

Como consequência desta mudança, novas regras de redução são definidas, como mostrado na Tabela 2.1. As 5 primeiras regras formam a relação $\to_x$, e o acréscimo da última forma a relação $\to_{Bx}$.

$x[x/u]$	$\rightarrow_{Var}\ u$	
$t[x/u]$	$\rightarrow_{Gc}\ t$	$se\ x \notin fv(t)$
$(t\ u)[x/v]$	$\rightarrow_{App}\ t[x/v]\ u[x/v]$	
$(\lambda y.\ u)[x/v]$	$\rightarrow_{Lamb}\ (\lambda y.\ u[x/v])$	$y \neq x, y \notin fv(u)$
$t[x/u][y/v]$	$\rightarrow_{Comp}\ t[y/v][x/u[y/v]]$	$se\ y\ \in fv(u)$
$(\lambda x.\ t)\ u$	$\rightarrow_{B}\ t[x/u]$	

Tabela 2.1: Regras de redução

Como estratégia para se obter a propriedade PSN, é adicionado ao sistema uma relação de equivalência, que permite a permutação de substituições independentes:

$$t[x/u][y/v] =_C t[y/v][x/u] \qquad se\ y \notin fv(u)\ \&\ x \notin fv(v)$$

A relação de equivalência $=_e$ é formada com a junção de $=_C$ e α-equivalência. Devido à mudança na gramática, precisamos estender a definição de α-equivalência para lidar com substituições explícitas.

Definição 2.2 (α-equivalência)**.** *Um termo $(\lambda x.t)$ é dito α-equivalente a $(\lambda y.u)$ se $t\{x/y\} = u$, com a ressalva que $y \notin fv(t)$ e $x \not\in fv(u)$. Um termo $t[x/u]$ é dito α-equivalente a $t'[y/u]$ se $t\{x/y\} = t'$. Mais geralmente, dois termos são ditos α-equivalentes se um pode ser obtido a partir do outro através de renomeamento de variáveis ligadas.*

Novamente, esta definição é uma relação de equivalência: para a reflexividade, basta fazer um renomeamento trivial. Para a transitivadade, basta compor os renomeamentos. Para a simetria, basta fazer o renomeamento contrário, ou seja: $u\{y/x\} = t$

As relações $\rightarrow_{ex}$ e $\rightarrow_{\lambda ex}$ são definidas como:

$$t\ \rightarrow_{ex}\ t' \iff \exists\ s, s'\ t.q.\ t\ =_e\ s\ \rightarrow_x\ s'\ =_e\ t'$$

$$t\ \rightarrow_{\lambda ex}\ t' \iff \exists\ s, s'\ t.q.\ t\ =_e\ s\ \rightarrow_{Bx}\ s'\ =_e\ t'$$

Estas relações serão amplamente utilizadas ao decorrer deste trabalho, cujo foco principal é contribuir com o andamento da formalização da propriedade **PSN** do sistema. A ideia da prova é definir uma estratégia de redução para este sistema, e utilizar esta estratégia para demonstrar que o conjunto dos termos fortemente normalizáveis do cálculo λ está contido no conjunto de termos fortemente normalizáveis do cálculo λex, garantindo a propriedade **PSN**.

Essencial para a prova da propriedade **PSN** é que a estratégia seja perpétua, definição que será apresentada posteriormente. Para isto, é necessário demonstrar a propriedade **IE** do sistema.

Definição 2.3 (Propriedade IE)**.** *Sejam t, u termos. Seja $SN_{\lambda ex}$ o conjunto de termos fortemente normalizáveis do sistema λex. Se $u \in SN_{\lambda ex}$ e $t\{x/u\} \in SN_{\lambda ex}$, então $t[x/u] \in SN_{\lambda ex}$.*

Esta propriedade pode ser entendida intuitivamente da seguinte maneira: Dados t, u tais que u e $t\{x/u\}$ são fortemente normalizáveis no sistema, então o correspondente utilizando substituições explícitas, $t[x/u]$ também será. Ou seja, a normalização da substituição implícita implica a normalização da explícita.

Para realizar a prova da propriedade **IE**, adicionamos um novo formalismo na gramática, *marcando* algumas substituições explícitas que sabemos que não introduzem problemas de normalização. Dividiremos, então, a regra principal deste sistema estendido, a relação $\lambda\underline{ex}$, em duas novas relações complementares: $\lambda\underline{ex}^i$ e $\lambda\underline{ex}^e$, chamadas reduções interna e externa, respectivamente. Estas reduções serão usadas para termos um melhor controle das reduções efetuadas dentro de substituições marcadas.

Com estes conceitos em mente, podemos definir de maneira mais clara a contribuição deste trabalho: continuar a formalização iniciada em (27), adicionando o formalismo de substituições marcadas, demonstrando que este preserva as propriedades já provadas para as substituições explícitas e, por fim, definir as reduções interna e externa acima mencionadas. Ou seja, o objetivo final é formalizar a equivalência $\lambda\underline{ex} = \lambda\underline{ex}^i \cup \lambda\underline{ex}^e$, chave para a prova da propriedade **IE**, permitindo, então, a conclusão da formalização da propriedade **PSN** do sistema λex.

2.1.1 A Gramática de pré-termos

Nesta seção, será feita um paralelo entre as estruturas básicas do sistema λex e sua formalização em Coq.

Como visto na Seção 2.1, o sistema possui uma gramática que pode ser vista como uma extensão do cálculo λ original, consistindo de variáveis, abstrações, aplicações e *substituições explícitas*. Ela pode ser descrita sucintamente como abaixo.

$$\tau := x \mid \lambda x.\tau \mid \tau\tau \mid \tau[x/\tau]$$

Já encontramos então nossa primeira divergência na formalização. Neste projeto, será usada a representação chamada *Locally Nameless Representation* (LNR), introduzida na subseção 1.2.2. Como visto anteriormente, existem termos nesta representação que não possuem correspondentes no sistema original, por conta da possibilidade de existirem índices que não estão ligados a nenhuma abstração.

Assim, torna-se novamente necessária uma gramática de *pré-termos*, que consiste de todas as expressões que podem escritas em LNR. Esta gramática é análoga à gramática

LNR apresentada em 1.2.2, porém agora adicionamos também o construtor de substituições explícitas.

$$\tau := x \mid n \mid \lambda\tau \mid \tau\tau \mid \tau[\tau]$$

Esta gramática é especificada usando um tipo indutivo.

```
Inductive pterm : Set :=
  | pterm_bvar : nat → pterm
  | pterm_fvar : var → pterm
  | pterm_app : pterm → pterm → pterm
  | pterm_abs : pterm → pterm
  | pterm_sub : pterm → pterm → pterm
```

Veja que variáveis livres e ligadas possuem construtores distintos. Para as ligadas, o construtor correto é `pterm_bvar`, que recebe um natural representando um índice. As variáveis livres são construídas com `pterm_fvar`, recebendo um elemento do tipo `var`, definido no framework de Charguéraud. As aplicações, abstrações e substituições são representadas através do construtores `pterm_app`, `pterm_abs` e `pterm_sub`, respectivamente.

Devemos também estender as operações de substituição e fechamento de pré-termos para se adequar à nova gramática, adicionando os seguintes casos:

$$\{k \to x\}(t[u]) \quad = \quad \{k+1 \to x\}t[\{k \to x\}u]$$

$$\{k \leftarrow x\}(t[u]) \quad = \quad \{k+1 \leftarrow x\}t[\{k \leftarrow x\}u]$$

2.2 Termos e Relações

2.2.1 Termos bem formados

Devido à correspondência assimétrica entre os termos do sistema λex e em LNR, torna-se necessário redefinir os predicados de boa formação de termos.

Como pré-requisito para a redefinição destes predicados, precisamos implementar as noções de abertura e fechamento de termos, como visto na subseção anterior. Relembrando, a operação de abertura é definida como $t^x = \{0 \to x\}t$.

Para implementar tal operação, precisaremos de duas funções, `open_rec` e `open`.

```
Fixpoint open_rec (k : nat) (u : pterm) (t : pterm) {struct t} : pterm := ...
```

```
Definition open t u := open_rec 0 u t.
Notation "{k ~> u} t":= (open_rec k u t) (at level 67).
Notation "t ^ x":= (open t (pterm_fvar x)).
```

A primeira adentra o termo `t` recursivamente, procurando pelo índice `k` e substituindo pelo termo `u`. Ao encontrar uma abstração ou substituição, o índice `k` é incrementado. A segunda, que é a chamada da operação de fato, apenas chama `open_rec` com $k = 0$.

Similarmente, a operação de fechamento é definida como $^{\backslash x}t \equiv \{0 \leftarrow t\}$. Esta operação é definida através das funções `close_rec` e `close`.

```
Fixpoint close_rec (k : nat) (x : var) (t : pterm) {struct t} : pterm := ...
Definition close t x := close_rec 0 x t.
```

Definição 2.4. *Sejam t, u pré-termos; L um conjunto finito de variáveis e x uma variável. Definimos o predicado term, que caracteriza termos bem formados, com base nas seguintes regras de inferência:*

$$\frac{}{term(x)}\ \text{TERM\_VAR} \qquad \frac{term\ t \qquad term\ u}{term(t\ u)}\ \text{TERM\_APP}$$

$$\frac{\forall x \notin L,\ term\ (t^x)}{term(\lambda t)}\ \text{TERM\_ABS} \qquad \frac{\forall x \notin L,\ term\ (t^x) \qquad term\ u}{term(t[u])}\ \text{TERM\_SUB}$$

O predicado *term* é o análogo do predicado *lc*, na definição 1.11, neste sistema. Ele recebe um pré-termo e indica que este elemento é bem formado, ou seja, não possui índices livres. Para cada construtor de pré-termo, temos um construtor diferente do predicado. Observe que não existe regra de inferência para índices livres, como desejado.

Em alguns casos, como dito na subseção 1.2.2, pode ser mais vantajoso verificar se um termo está bem formado com uma função recursiva. Tal verificação é feita através do predicado `term'`, que deve funcionar de maneira equivalente ao predicado `term`. Este novo predicado é definido com base na definição recursiva `lc_at`, que verifica se o termo t está *fechado* a um nível k, ou seja, se não existe índice livre de valor maior ou igual a k. Sua implementação é feita com base na Definição 1.12, adicionando apenas o caso a seguir, para substituições explícitas.

$$lc\_at\ k\ (t1[t2]) \quad \equiv \quad (lc\_at\ (Sk)\ t1)\ \&\ (lc\_at\ k\ t2)$$

```
Fixpoint lc_at (k:nat) (t:pterm) {struct t} : Prop := ...
Definition term' t := lc_at 0 t.
```

Assim, essa equivalência entre os predicados *term* e *term'* também teve que ser formalizada, utilizando uma série de lemas auxiliares.

Primeiramente, foram provados dois resultados para o predicado `lc_at`. Análogos ao Lema 1.1.

```
Lemma lc_rec_open_var_rec : ∀ x t k, lc_at k (open_rec k x t) → lc_at (S k) t.
```

Este lema garante que, se um termo t, aberto com uma variável a um nível k, é fechado a este mesmo nível k, então o termo t sem a abertura é fechado a nível $k + 1$.

```
Lemma lc_at_open_var_rec : ∀ x t k, lc_at (S k) t → lc_at k (open_rec k (pterm_fvar
x) t).
```

O segundo é recíproco: Se um termo t é fechado a um nível $k + 1$, então este mesmo termo, aberto com uma variável a nível k, é fechado a nível k.

Já podemos agora provar a equivalência entre as duas definições de termos bem formados. Este resultado é o análogo em nossa formalização do Teorema 1.

```
Lemma term_eq_term' : ∀ t, term t ↔ term' t.
```

Demonstração.

(→) Este caso é bem direto, fazendo uma indução no predicado *term* t. As hipóteses de indução resolvem os casos diretamente, exceto para abstração e substituição. Nesses, a hipótese é dada para t^x, quando precisamos provar $lc\_at\ 1\ t$. Felizmente os lemas de enfraquecimento acima resolvem, bastando usar o lema *lc_rec_open_var_rec*. Observando este lema, vemos que podemos concluir $lc\_at\ 1\ t$ se tivermos $lc\_at\ 0\ (open\_rec\ 0\ x\ t)$. Ou seja, se tivermos $lc\_at\ 0\ t^x$, que é nossa hipótese de indução.

(←) O segundo caso não é tão imediato, pois fazemos uma indução no *tamanho do termo*, para auxiliar na prova. A hipótese de indução se refere a termos com tamanho igual ao do termo sobre o qual queremos provar a propriedade. Isto é útil pois o tamanho de um termo ao ser aberto com uma variável não muda. Novamente, no caso da abstração e substituição, precisamos fazer um ajuste utilizando o lema *lc_at_open_var_rec*. Queremos provar *term* $(pterm\_abs\ t1)$. Temos como hipótese que vale $lc\_at\ 1\ t1$ e que, para todo termo $t2$, de mesmo tamanho que o termo $t1$, e

toda variável x que não ocorre livre em $t2$, se vale $lc\_at\ 0\ t2^x$, vale $term\ (t2^x)$. Assim, analisando o construtor de $term$ para abstração e escolhendo o conjunto L como $fv(t1)$, precisamos provar que, para qualquer $x \notin fv(t1)$, vale $term\ (t1^x)$. Assim, podemos usar a hipótese de indução, bastando provar agora que vale $lc\_at\ 0\ t1^x$. Observando o lema $lc\_at\_open\_var\_rec$, vemos que basta mostrar que vale $lc\_at\ 1\ t1$, que é dado como hipótese. A prova é similar para o caso da substituição.

□

Com esta prova, fica mais evidente as dificuldades que surgem no processo de formalização. Enquanto a prova do Teorema 1 é bem direta, a nossa versão tem que lidar com vários detalhes explicitamente, e em um dos casos tem que adotar até mesmo outra estratégia de prova, fazendo indução no tamanho do termo.

2.2.2 Equivalência de termos

Existem várias maneiras de se obter a preservação da normalização forte em um cálculo com substituições explícitas (26). No caso do sistema λex, a estratégia é adicionar uma relação de equivalência entre os termos.

Para isto, observemos o lema da substituição, no cálculo λ:

$$t\{x/u\}\{y/v\} = t\{y/v\}\{x/u\}, \quad x \notin fv(v),\ y \notin fv(u)$$

Basicamente, se as substituições são *independentes*, podemos permutá-las sem alterar seu resultado. Queremos que o análogo para substituições explícitas também seja válido, ou seja, que $t[x/u][y/v] \equiv t[y/v][x/u]$, se $x \notin fv(v)$ e $y \notin fv(u)$. Porém, os dois termos são sintaticamente distintos. Para resolver este problema, adicionamos no sistema uma regra de *permutação de substituições independentes*.

$$t[x/u][y/v] =_C t[y/v][x/u] \qquad se\ y \notin fv(u)\ \&\ x \notin fv(v)$$

Dizemos que ambas as representações são *equivalentes*. Precisamos, então, formalizar essa definição no sistema.

Na formalização abaixo, a ocorrência do operador & troca todas as ocorrências de índices zero e um, para que eles continuem se referindo à mesma substituição.

```
Inductive eqc : pterm → pterm → Prop :=
  | eqc_def: ∀ t u v, lc_at 2 t → term u → term v → eqc (t[u][v]) ((& t)[v][u]).
```

Veja que exigimos que u e v sejam termos, e que deva valer `lc_at 2 t`. Isto é necessário para que eqc seja equivalente à equação $=_C$. Na definição original, queremos que $y \notin fv(u)$

e $x \notin fv(v)$, para que as substituições sejam independentes. Aqui as exigências *term u* e *term v* cumprem esse papel, mas são, também, mais fortes: queremos trabalhar apenas com termos sem índices livres. Este fato também justifica a exigência de que `lc_at 2 t`.

Note que esta definição ainda possui muitas limitações. Por exemplo, se um termo possui uma lista de substituições, só podemos trocar as duas últimas. Também não é possível realizar permutações em subtermos, ou várias permutações seguidas. Assim, precisamos criar *fechos* em cima dessa definição, para ajustar às nossas necessidades.

Definição 2.5 (Fecho contextual)**.** *Sejam t, t', u pré-termos; x uma variável e R uma relação binária entre pré-termos. As regras a seguir definem o fecho contextual de R.*

$$\frac{(R\ t\ t')}{(ES\_contextual\_closure\ R)\ t\ t'}\ \text{ES\_REDEX}$$

$$\frac{((ES\_contextual\_closure\ R)\ t\ t'),\ (term\ u)}{((ES\_contextual\_closure\ R)\ (t\ u)\ (t'\ u))}\ \text{ES\_APP\_LEFT}$$

$$\frac{((ES\_contextual\_closure\ R)\ t\ t'),\ (term\ u)}{((ES\_contextual\_closure\ R)\ (u\ t)\ (u\ t'))}\ \text{ES\_APP\_RIGHT}$$

$$\frac{(\forall\ x,\ x \notin L \to ((ES\_contextual\_closure\ R)\ t^x\ t'^x))}{((ES\_contextual\_closure\ R)\ (\lambda.t)\ (\lambda.t'))}\ \text{ES\_ABS\_IN}$$

$$\frac{(\forall\ x,\ x \notin L \to ((ES\_contextual\_closure\ R)\ t^x\ t'^x)),\ (term\ u)}{((ES\_contextual\_closure\ R)\ (t[u])\ (t'[u]))}\ \text{ES\_SUBST\_LEFT}$$

$$\frac{((ES\_contextual\_closure\ R)\ u\ u'),\ (body\ t)}{((ES\_contextual\_closure\ R)\ (t[u])\ (t[u']))}\ \text{ES\_SUBST\_RIGHT}$$

```
Definition eqc_ctx (t u: pterm) := ES_contextual_closure eqc t u.
Notation "t =c u":= (eqc_ctx t u) (at level 66).
```

A definição `ES_contextual_closure` é um dos chamados *fechos contextuais*. A ideia é que, se vale $t \to_R t'$, então, para qualquer termo construído a partir de t, vale a redução pelo fecho contextual de R, para um termo análogo, mas construído a partir de t'. Como exemplo:

$$(t \to_R t') \Rightarrow ((pterm\_app\ t\ u) \to_{ES\_Contextual\_Closure\ R} (pterm\_app\ t'\ u))$$

Definimos então o análogo da equação $=_C$, com base no fecho transitivo-reflexivo da relação `eqc_ctx`

```
Definition eqc_trans (t u: pterm) := trans_closure eqc_ctx t u.
Notation "t =c+ u":= (eqc_trans t u) (at level 66).
Definition eqC (t : pterm) (u : pterm) := star_closure eqc_ctx t u.
Notation "t =e u":= (eqC t u) (at level 66).
```

Assim, a relação de equivalência que correspondente à relação $=_C$ é a `=e`, que permite vários passos de permutações, além de permutações no interior dos termos. Uma importante diferença de provas usando equivalências no papel e numa formalização é que resultados que são intuitivos e admitidos num ambiente informal devem ser provados minuciosamente.

Como exemplo, podemos querer mostrar a compatibilidade da igualdade com a função `lc_at`. Isto é intuitivo, pois permutar duas substituições não irá criar índices livres. Porém, é preciso entrar em detalhes na prova formal.

Observe que precisamos provar três resultados. Não basta mostrar que o predicado `eqc` é compatível com `lc_at`, pois ainda usaremos seus fechos.

```
Lemma lc_at_eqc : ∀ n t u, eqc t u → (lc_at n t ↔ lc_at n u).
```

A prova de `lc_at_eqc` já apresenta certos detalhes. Ela é feita através de uma análise de casos do predicado `eqc`. É preciso realizar certos ajustes com índices devido ao uso do operador `&` na definição da equação. Usamos também a equivalência entre `lc_at` e o predicado `term` para usar a hipótese que as expressões dentro das substituições são termos, além de regras de enfraquecimento para o `lc_at`.

```
Lemma lc_at_ES_ctx_eqc : ∀ n t u, (ES_contextual_closure eqc) t u → (lc_at n t ↔
lc_at n u).
```

No caso do fecho contextual, fazemos indução no próprio fecho. As hipóteses de indução resolvem os casos mais simples. Os casos de abstração e substituição se tornam um pouco mais complexos, pois temos que lidar com o corpo dos pré-termos. A hipótese de indução é dada da seguinte maneira: dado um conjunto L, para toda variável x que não está em L, e todo natural n, vale $lc\_at\ n\ (t^x) \iff lc\_at\ n\ (t'^x)$. Precisamos, porém, provar que vale $lc\_at\ (S\ n)\ t \iff lc\_at\ (S\ n)\ t'$. Para resolver este problema tomamos uma variável qualquer que não esteja em L, e abrimos os subtermos t e t' com ela, a um nível zero. Podemos fazer isto sem afetar a validade do predicado lc_at, pois não estamos adicionando nenhum novo índice. O objetivo então se torna provar

$lc\_at\ (S\ n)\ (t^x) \iff lc\_at\ (S\ n)\ (t'^x)$ Basta então aplicar a hipótese de indução, concluindo, assim, a prova.

```
Lemma lc_at_eqC : ∀ n t t', t =e t' → (lc_at n t ↔ lc_at n t').
```

Esta prova é bem simples, fazendo indução no fecho transitivo-reflexivo. O caso reflexivo é trivial. No passo indutivo, fechamos facilmente usando a hipótese de indução, pela transitividade da relação $\leftrightarrow$.

Esta é uma parte do trabalho especialmente extensa e detalhada. É preciso mostrar que vários predicados importantes da teoria são preservados pelas classes de equivâlencia entre termos. Várias vezes durante as provas trabalharemos com termos equivalentes módulo $=_e$ e, sem uma base bem construída de resultados sobre a relação `eqC`, fica impossível concluí-las.

Os principais resultados que foram provados nesta parte incluem:

- Preservação da estrutura de termos pela equivalência. Construtores de termos, abertura de termos, substituições e renomeamento de variáveis devem funcionar de maneira análoga para dois termos equivalentes.
- Boa formação de corpos de abstrações, conjunto de variáveis livres e reduções dentro de fechos contextuais também devem ser preservadas pela equivalência.

2.2.3 Reduções do Sistema

Com a estrutura de termos e as regras de equivalência bem estabelecidas, podemos iniciar a formalização das reduções do sistema λex. As regras de redução listadas em 2.1 serão formalizadas nos tipos indutivos `sys_x` e `rule_b`.

```
Inductive rule_b : pterm → pterm → Prop :=
    reg_rule_b : ∀ (t u : pterm), body t → term u →
      rule_b (pterm_app (pterm_abs t) u) (t[u]).
Notation "t ->_B u":= (rule_b t u) (at level 66).

Inductive sys_x : pterm → pterm → Prop :=
| reg_rule_var : ∀ t, term t → sys_x (pterm_bvar 0 [t]) t
| reg_rule_gc : ∀ t u, term t → term u → sys_x (t[u]) t
| reg_rule_app : ∀ t1 t2 u, body t1 → body t2 → term u →
  sys_x ((pterm_app t1 t2)[u]) (pterm_app (t1[u]) (t2[u]))
| reg_rule_lamb : ∀ t u, body (pterm_abs t) → term u →
  sys_x ((pterm_abs t)[u]) (pterm_abs ((& t)[u]))
| reg_rule_comp : ∀ t u v, body (t[u]) → ¬ term u → term v →
```

```
  sys_x (t[u][v]) (((& t)[v])[ u[ v ] ]).
Notation "t ->_x u":=
  (sys_x t u) (at level 59, left associativity).
```

A relação $\to_x$ é definida como sendo exatamente o predicado `sys_x`. Algumas exigências técnicas de *term* e *body* são adicionadas para auxiliar nas provas, já que queremos sempre trabalhar com termos bem formados.

O predicado `rule_b` é o que formaliza a regra $\to_B$, sendo esta a regra que reduz uma aplicação a uma substituição explícita. Podemos, a partir destes dois predicados, definir a regra de redução principal do sistema λex.

```
Inductive sys_Bx: pterm → pterm → Prop :=
| B_lx : ∀ t u, t →_B u → sys_Bx t u
| ∀ t u, t →_x u → sys_Bx t u.
Notation "t ->_Bx u":= (sys_Bx t u) (at level 59, left associativity).
Definition red_ctx_mod_eqC (R: pterm → pterm → Prop) (t: pterm) (u : pterm) :=
           ∃ t', ∃ u', (t =e t')/\(ES_contextual_closure R t' u')/\(u' =e u).
Definition lex t u := red_ctx_mod_eqC sys_Bx t u.
Notation "t –>lex u":= (lex t u) (at level 66).
Definition lex_trs t u := trans_closure lex t u.
Notation "t –>lex+ u":= (lex_trs t u) (at level 66).
Definition lex_str t u := star_closure lex t u.
Notation "t –>lex* u":= (lex_str t u) (at level 66).
```

O análogo da regra $\to^*_{\lambda ex}$ no sistema será a relação `"->lex*"`, construída a partir do fecho transitivo-reflexivo, contextual e equacional da relação $\to_{Bx}$.

Para auxiliar nas provas, definimos alguns predicados que garantem que estamos trabalhando sempre com termos, e que representam algumas noções intuitivas das reduções.

Definição 2.6 (Regularidade)**.** *Sejam t, u pré-termos e R uma relação binária entre pré-termos. Dizemos que R é regular se, sempre que vale $R\ t\ u$, então t e u são termos bem formados. A noção de regularidade é formalizada pelo predicado red_regular, definido como base na regra de inferência abaixo.*

$$\frac{(red\_regular\ R),\ (R\ t\ u)}{term\ t\ \wedge\ term\ u}$$

A propriedade de regularidade é interessante para facilitar diversas provas, já que desejamos evitar trabalhar com termos que possuem índices livres. Basicamente, se fizemos a redução R entre dois pré-termos, ambos são termos bem formados.

Algumas relações interessantes não são capazes de garantir a regularidade. Ainda assim, queremos continuar lidando com termos bem formados. Definimos então uma noção mais fraca de regularidade, que apenas preserva a boa formação de termos.

Definição 2.7 (Regularidade fraca)**.** *Sejam t, u pré-termos e R uma relação binária entre pré-termos. Dizemos que R satisfaz a regularidade fraca se, sempre que vale $R\ t\ u$, vale $(term\ t) \iff (term\ u)$. A noção de regularidade fraca é formalizada pelo predicado $red\_regular'$, definido como base na regra de inferência abaixo.*

$$\frac{(red\_regular'\ R),\ (R\ t\ u)}{term\ t \iff term\ u}$$

Definição 2.8 (Renomeamento)**.** *Sejam t, u pré-termos; x, y variáveis e R uma relação binária entre pré-termos. Dizemos que a relação R é compatível com renomeamento de variáveis, ou seja, vale $red\_rename\ R$, se R satisfaz a seguinte regra de inferência:*

$$\frac{(red\_rename\ R),\ (\forall x,\ x \notin fv(t) \to R\ (t^x)\ (u^x))}{R\ (t^y)\ (u^y)}$$

Basicamente, se um termo t aberto com uma variável $x \notin fv(t)$ se reduz a um t', também aberto com x, então este x pode ser trocado por outra variável $y \notin fv(t)$, preservando a redução $\to_R$.

Estes resultados auxiliam muito em diversas provas. Em especial, os resultados envolvendo abertura de termos e índices livres são necessários para provas em que fazemos indução na estrutura do termo, pois, em geral, no caso da abstração as hipóteses se referem ao sub-termo aberto com uma variável.

2.3 Preservação da normalização forte

Nesta seção, queremos dar uma visão geral da propriedade PSN e descrever a formalização da propriedade IE, foco deste trabalho.

A propriedade PSN é a que garante que, se um termo t é fortemente normalizável no cálculo original, ou seja, toda cadeia de reduções a partir dele é finita, então ele também é fortemente normalizável no sistema λex.

A prova da PSN é feita primeiramente definindo uma estratégia de redução *perpétua* para o sistema.

Definição 2.9 (Estratégia de redução perpétua)**.** *Uma estratégia perpétua fornece uma cadeia de reduções* infinita *para um termo, se uma existe. Caso não exista, fornece uma cadeia que termina em um termo em forma normal.*

Observe que, se um termo t não é fortemente normalizável, então a estratégia o reduzirá para um termo t' que também não o é, e assim por diante, para poder criar a cadeia de redução infinita. Em outras palavras: Se $t \to t'$ por uma estratégia perpétua, e se t não é fortemente normalizável, então t' também não será fortemente normalizável. Pela contrapositiva, se t se reduz, por esta estratégia, a um termo t' que é fortemente normalizável, então t também o deve ser.

Para a prova da propriedade **PSN**, observamos o uso desta estratégia para um caso particular: se um termo $t[x/u]$ é reduzido, por uma estratégia perpétua, para um termo $t\{x/u\}$, sendo este fortemente normalizável, então o termo original também será fortemente normalizável. Em outras palavras, a normalização da substituição *implícita* implica na normalização da substituição *explícita*. Esta é a chamada propriedade **IE**.

Para a prova da propriedade **IE**, é adicionado mais uma estrutura no sistema, chamada de *substituição marcada*. Esta é a estratégia apresentada em (26), e seguiremos a mesma linha nesta formalização.

2.3.1 Substituições marcadas

A ideia é controlar as reduções feitas envolvendo substituições explícitas. Para isto, adicionamos um novo construtor na gramática.

$$\tau := x \mid \lambda x.\tau \mid \tau\tau \mid \tau[x/\tau] \mid \tau[\![x/u]\!]$$

Na nova substituição, não podemos colocar qualquer qualquer termo no lugar de u. Restringimos o termo u a apenas termos da gramática original, ou seja, sem substituições marcadas. Além disso, é necessário que o termo seja fortemente normalizável. Para fazer esta verificação, usaremos o predicado `SN`, definido com base no predicado `SN_ind`.

Definição 2.10. *Seja t um pré-termo e R uma relação binária entre pré-termos. Dizemos que t é fortemente normalizável pela redução R, ou seja, vale (SN R t) se existe $n \in \mathbb{N}$ satisfazendo o predicado SN_ind, definido com base na regra de inferência abaixo.*

$$\frac{(\forall t', t \to_R t' \Rightarrow \exists k \in \mathbb{N}, k < n, (SN\_ind\ k\ t'))}{SN\_ind\ n\ t}\ \text{SN\_IND}$$

Essencialmente, $(SN\ R\ t)$ garante que existe $n \in \mathbb{N}$ tal que toda cadeia de reduções a partir de t tem comprimento no máximo n, ou seja, é finita.

Precisamos adicionar a nova substituição na nossa formalização. Adicionamos um novo construtor ao tipo `pterm`.

```
Inductive pterm : Set :=
  | ...
  | pterm_lsub : pterm → pterm → pterm.
```

Assim como precisamos de um predicado para verificar se um termo comum está bem formado, vamos criar um outro predicado, chamado `lab_term`.

Definição 2.11 (Termo marcado)**.** *Este predicado é uma extensão do predicado term, visto na Definição 2.4. Adicionamos apenas um caso para lidar com as substituições marcadas, sendo todos os outros casos análogos ao anterior.*

$$\frac{\forall x \notin L,\ lab\_term(t^x) \qquad term(u) \qquad (SN\ lex)\ u}{lab\_term(t[\![u]\!])}\ \text{LAB\_TERM\_SUB'}$$

Observe que no caso da substituição marcada, temos a exigência $(SN\ lex\ u)$, que indica que u é fortemente normalizável.

Também devemos estender a noção equivalente de ser localmente fechado, para manter a equivalência definida no caso do sistema sem as substituições marcadas. Assim, definimos uma nova função `lc_at'`, análoga à do caso não marcado. Sua implementação é também feita com base na Definição 1.12, com apenas uma extensão para os casos das substituição marcadas e explícitas.

Definição 2.12. *A função lc_at' é definida como uma extensão de lc_at, adicionando os seguintes dois casos:*

$$\begin{aligned} lc\_at'\ k\ (t[u]) &\equiv (lc\_at'\ (Sk)\ t)\ \&\ (lc\_at'\ k\ u) \\ lc\_at'\ k\ (t[\![u]\!]) &\equiv (lc\_at'\ (Sk)\ t)\ \&\ (lc\_at\ k\ u)\ \&\ (SN\ lex\ u) \end{aligned}$$

```
Fixpoint lc_at' (k:nat) (t:pterm) {struct t} : Prop := ...
Definition term'' t := lc_at' 0 t.
```

Como no capítulo anterior, precisamos mostrar a equivalência entre `lab_term` e `lc_at'`. Para isto, precisamos mostrar também os lemas que falam sobre a interação entre `lc_at'`

e a operação de substituição. As provas seguem de maneira análoga às suas versões originais. Devemos apenas tomar cuidado com as exigências de normalização nos casos das substituições marcadas. Para resolvê-los tivemos que assumir alguns resultados envolvendo a relação entre o predicado SN e índices livres.

Como no caso do sistema simples, queremos definir classes de equivalências de termos, para trabalhar módulo permutação de substituições. Para isso, precisamos definir novos fechos contextuais para os termos com substituições marcadas.

O fecho contextual para termos marcados é definido como uma extensão do fecho para termos comuns, visto na definição 2.5, com a ressalva que os predicados de *term* e *body* são substituídos por seus análogos, *lab_term* e *lab_body*, respectivamente. Os casos que lidam com substituições marcadas são definidos pelas regras de inferência a seguir.

$$\frac{(\forall\ x,\ x \notin L \to ((lab\_contextual\_closure\ R)\ t^{x}\ t'^{x})),\ (term\ u),\ (SN\ lex\ u)}{((lab\_contextual\_closure\ R)\ (t[\![u]\!])\ (t'[\![u]\!]))}\ \textsc{lab\_subst'\_left}$$

$$\frac{(R\ u\ u'),\ (lab\_body\ t)}{((lab\_contextual\_closure\ R)\ (t[\![u]\!])\ (t[\![u']\!]))}\ \textsc{lab\_subst'\_right}$$

São definidos também dois outros fechos análogos, `simpl_lab_contextual_closure` e `ext_lab_contextual_closure`. No primeiro, a diferença é que não é feita a exigência $(SN\ lex\ u)$ no caso da redução à esquerda da substituição marcada. Ou seja, temos:

$$\frac{(\forall\ x,\ x \notin l \to ((simpl\_lab\_contextual\_closure\ R)\ t^{x}\ t'^{x})),\ (term\ u)}{((simpl\_lab\_contextual\_closure\ R)\ (t[\![u]\!])\ (t'[\![u]\!]))}\ \textsc{simpl\_lab\_subst'\_left}$$

$$\frac{(r\ u\ u'),\ (lab\_body\ t)}{((simpl\_lab\_contextual\_closure\ R)\ (t[\![u]\!])\ (t[\![u']\!]))}\ \textsc{simpl\_lab\_subst'\_right}$$

No segundo, queremos reduzir apenas fora de substituições marcadas. Assim, o fecho não possui um análogo ao caso `SIMPL_LAB_SUBST'_RIGHT`.

$$\frac{(\forall\ x,\ x \notin L \to ((ext\_lab\_contextual\_closure\ R)\ t^{x}\ t'^{x})),\ (term\ u)}{((ext\_lab\_contextual\_closure\ R)\ (t[\![u]\!])\ (t'[\![u]\!]))}\ \textsc{ext\_lab\_subst'\_left}$$

Podemos então definir a relação equacional para termos marcados.

```
Inductive lab_eqc : pterm → pterm → Prop :=
  | lab_eqc_rx1 : ∀ t u v,
    lab_term u → term v → lab_eqc (t[u][[v]]) ((& t)[[v]][u])
  | lab_eqc_rx2 : ∀ t u v,
    term u → lab_term v → lab_eqc (t[[u]][v]) ((& t)[v][[u]])
  | lab_eqc_rx3 : ∀ t u v,
    term u → term v → lab_eqc (t[[u]][[v]]) ((& t)[[v]][[u]]).
```

Os construtores basicamente definem como permutar duas substituições, desde que uma delas seja marcada. Como requisito, é necessário garantir a propriedade `lab_term` ou `term`, dependendo da substituição. A ideia é permitir que as substituições marcadas sejam permutadas "para dentro" do termo, passando por uma substituição comum ou marcada.

A equação principal utilizada nos termos marcados será a relação $=_{\underline{e}}$, formalizada como o fecho contextual e transitivo do predicado `lab_eqc`.

```
Definition lab_eqC (t: pterm) (u : pterm) := trans_closure (lab_contextual_closure
lab_eqc) t u .
Notation "t =~e u":= (lab_eqC t u) (at level 66).
```

Também é preciso definir um predicado análogo de regularidade para termos marcados.

Definição 2.13 (Regularidade para termos marcados)**.** *Sejam t, u pré-termos e R uma relação binária entre pré-termos. Dizemos que R é regular se, sempre que vale $R\ t\ u$, então t e u são termos marcados bem formados. A noção de regularidade para termos marcados é formalizada pelo predicado red_lab_regular, definido com base na regra de inferência abaixo.*

$$\frac{(red\_lab\_regular\ R),\ (R\ t\ u)}{lab\_term\ t\ \wedge\ lab\_term\ u}$$

Definição 2.14 (Regularidade fraca para termos marcados)**.** *Sejam t, u pré-termos e R uma relação binária entre pré-termos. Dizemos que R satisfaz a regularidade fraca se, sempre que vale $R\ t\ u$, vale $(lab\_term\ t) \iff (lab\_term\ u)$. A noção de regularidade fraca é formalizada pelo predicado red_lab_regular′, definido com base na regra de inferência abaixo.*

$$\frac{(red\_lab\_regular'\ R),\ (R\ t\ u)}{lab\_term\ t\ \iff\ lab\_term\ u}$$

Com todas as estruturas e propriedades para termos marcados bem definidos, podemos seguir com a prova da propriedade IE.

2.3.2 Equivalência de reduções com o sistema original

Queremos utilizar esse sistema estendido com as substituições marcadas para estudar o sistema original. Para isto, precisamos estender a regra de redução do sistema para lidar com as novas substituições. Definimos então a redução $\to_{\underline{x}}$, como em (26), na tabela 2.2.

$x\llbracket x/u\rrbracket$	$\to_{Var}\ u$	
$t\llbracket x/u\rrbracket$	$\to_{Gc}\ t$	$se\ x \notin fv(t)$
$(t\ u)\llbracket x/v\rrbracket$	$\to_{App}\ t\llbracket x/v\rrbracket\ u\llbracket x/v\rrbracket$	
$(\lambda y.\ u)\llbracket x/v\rrbracket$	$\to_{Lamb}\ (\lambda y.\ u\llbracket x/v\rrbracket)$	
$t\llbracket x/u\rrbracket\llbracket y/v\rrbracket$	$\to_{Comp}\ t\llbracket y/v\rrbracket\llbracket x/u\llbracket y/v\rrbracket\rrbracket$	$se\ y\ \in fv(u)$

Tabela 2.2: A redução $\to_{\underline{x}}$

Assim, a relação $\to_{\lambda\underline{ex}}$ é definida como a união das reduções $\to_{Bx}$ e $\to_{\underline{x}}$, módulo $=_\alpha$, $=_e$ e $=_{\underline{e}}$:

$$t \to_{\lambda\underline{ex}} t' \iff \exists s,\ s';\ t =_{e\cup\underline{e}\cup\alpha} s \to_{Bx\cup\underline{x}} s' =_{e\cup\underline{e}\cup\alpha} t'$$

Para provar a propriedade PSN, será necessário relacionar a redução $\to_{\lambda\underline{ex}}$ com a redução original, $\to_{\lambda ex}$. Para isto, iremos decompor a redução em termos marcados em duas novas reduções, $\to_{\lambda\underline{ex}^i}$ e $\to_{\lambda\underline{ex}^e}$, que também agem em termos marcados.

Definição 2.15 (Redução interna)**.** *A relação $\lambda\underline{ex}^i$, chamada de* redução interna*, é definida adicionando à redução $\to_{\underline{ex}}$ a redução $\to_{\lambda ex}$ no corpo das substituições marcadas. Formalmente, a relação $\to_{\lambda\underline{ex}^i}$ é definida como a seguinte redução, $\to_{\lambda\underline{x}^i}$, módulo $=_\alpha$, $=_e$ e $=_{\underline{e}}$:*

- *Se $u \to_{Bx} u'$ e $u,\ u'$ são termos, então $t\llbracket x/u\rrbracket \to_{\lambda\underline{x}^i} t\llbracket x/u'\rrbracket$*
- *Se $t \to_{\underline{x}} t'$, então $t \to_{\lambda\underline{x}^i} t'$*
- *Se $t \to_{\lambda\underline{x}^i} t'$, então vale $t\ u \to_{\lambda\underline{x}^i} t'\ u$, $u\ t \to_{\lambda\underline{x}^i} u\ t'$, $\lambda x.t \to_{\lambda\underline{x}^i} \lambda x.t'$, $t[x/u] \to_{\lambda\underline{x}^i} t'[x/u]$, $u[x/t] \to_{\lambda\underline{x}^i} u[x/t']$ e $t\llbracket x/u\rrbracket \to_{\lambda\underline{x}^i} t'\llbracket x/u\rrbracket$.*

Definição 2.16 (Redução externa)**.** *A relação* $\lambda\underline{ex}^e$*, chamada de* redução externa*, é definida como a redução* λex *em todos os lugares de um termo,* exceto *no corpo das substituições marcadas. Formalmente, a relação* $\rightarrow_{\lambda\underline{ex}^e}$ *é definida como a seguinte redução,* $\rightarrow_{\lambda\underline{x}^e}$*, módulo* $=_\alpha$*,* $=_e$ *e* $=_{\underline{e}}$*:*

- *Se* $t \rightarrow_{Bx} t'$ *ocorre fora de uma substituição marcada, então* $t \rightarrow_{\lambda\underline{x}^e} t$
- *Se* $t \rightarrow_{\lambda\underline{x}^e} t'$*, então vale* $t\ u \rightarrow_{\lambda\underline{x}^e} t'\ u$*,* $u\ t \rightarrow_{\lambda\underline{x}^e} u\ t'$*,* $\lambda x.t \rightarrow_{\lambda\underline{x}^e} \lambda x.t'$*,* $t[x/u] \rightarrow_{\lambda\underline{x}^e} t'[x/u]$*,* $u[x/t] \rightarrow_{\lambda\underline{x}^e} u[x/t']$ *e* $t[\![x/u]\!] \rightarrow_{\lambda\underline{x}^e} t'[\![x/u]\!]$*.*

O objetivo principal deste trabalho será a formalização destas duas reduções e a prova de equivalências da união destas com a redução "`-->[lex]`", que é a redução $\rightarrow_{\lambda\underline{ex}}$ formalizada. Em outras palavras, queremos provar o seguinte Teorema:

Teorema 2 (Equivalência entre as reduções)**.** *Seja* t *um termo bem formado, podendo possuir substituições marcadas. Então vale que* $t \rightarrow_{\lambda\underline{ex}} t' \iff t \rightarrow_{\lambda\underline{ex}^i \cup \lambda\underline{ex}^e} t'$*.*

Para iniciar a formalização, definimos mais uma equação, $=_{EE}$, que serve como união de ambas as equações anteriores, $=_e$ e $=_{\underline{e}}$:

```
Definition eqcc t t' := eqc t t' ∨ lab_eqc t t'.
Notation "t =ee t'":= (eqcc t t') (at level 66).

Definition star_ctx_eqcc (t: pterm) (u : pterm) := star_closure (simpl_lab_contextual_closure
eqcc) t u .
Notation "t =EE u":= (star_ctx_eqcc t u) (at level 66).
```

Como primeiro passo para a formalização deste teorema, devemos definir predicados análogos às reduções $\lambda\underline{ex}$, $\lambda\underline{ex}^i$, $\lambda\underline{ex}^e$.

Iniciamos, então, definindo a relação $\rightarrow_{\underline{x}}$, dada pela tabela 2.2, como o tipo `lab_sys_x`:

```
Inductive lab_sys_x : pterm → pterm → Prop :=
| lab_reg_rule_var : ∀ t, lab_term (pterm_bvar 0 [[t]]) → lab_sys_x (pterm_bvar 0 [[t]]) t
| lab_reg_rule_gc : ∀ t u, lab_term t → lab_term (t[[u]]) → lab_sys_x (t[[u]]) t
| lab_reg_rule_app : ∀ t1 t2 u, lab_term (t1[[u]]) → lab_term (t2[[u]]) →
  lab_sys_x ((pterm_app t1 t2)[[u]]) (pterm_app (t1[[u]]) (t2[[u]]))
| lab_reg_rule_lamb : ∀ t u, lab_term ((pterm_abs t)[[u]]) →
  lab_sys_x ((pterm_abs t)[[u]]) (pterm_abs ((& t)[[u]]))
| lab_reg_rule_comp : ∀ t u v, lab_term ((t[u])[[v]]) → ¬ term u →
  lab_sys_x (t[u][[v]]) (((& t)[[v]])[u[[v]]]).

Notation "t ->_lab_x u":= (lab_sys_x t u) (at level 59, left associativity).
```

Agora já podemos definir o análogo da relação $\lambda\underline{ex}$. Para isto, começamos com a formalização da relação $\rightarrow_{Bx \cup \underline{x}}$, no tipo indutivo `lab_sys_lx`:

```
Inductive lab_sys_lx: pterm → pterm → Prop :=
| B_lx : ∀ t u, t →_B u → lab_sys_lx t u
| sys_x_lx : ∀ t u, t →_x u → lab_sys_lx t u
| sys_x_lab_lx : ∀ t u, t →_lab_x u → lab_sys_lx t u.
```

Assim, a especificação da relação $\lambda\underline{ex}$ será `"-->[lex]"`, dado por:

```
Definition lab_lex (t: pterm) (u : pterm) :=
    ∃ t' u', (t =EE t')/\(lab_contextual_closure lab_sys_lx t' u')/\(u' =EE u).
Notation "t –>[lex] u":= (lab_lex t u) (at level 59, left associativity).
```

Em seguida, iremos definir a redução externa. Ela consistirá apenas da aplicação da regra do sistema original, $\rightarrow_{Bx}$, em qualquer parte de um termo que seja fora de uma substituição explícita. Em outras palavras, ela será o fecho contextual externo da relação $\rightarrow_{Bx}$, módulo $=_{EE}$.

```
Notation "t –>[lx_e] u":= (ext_lab_EE_ctx_red sys_Bx t u) .
```

O fecho `ext_lab_EE_ctx_red` é o que realiza uma relação em qualquer ponto de um termo, **exceto** dentro de uma substituição marcada, permitindo a permutação de substituições tanto antes quanto depois da redução ser feita. Ele realiza isto utilizando o fecho `ext_lab_contextual_closure` e aplicando a equação `=EE`.

```
Definition ext_lab_EE_ctx_red (R: pterm → pterm → Prop) (t: pterm) (u : pterm) :=
∃ t' u', (t =EE t')/\(ext_lab_contextual_closure R t' u')/\(u' =EE u).
```

Para definir a redução interna, precisamos do predicado `lab_x_i`, que irá permitir aplicações de $\rightarrow_{Bx}$ apenas dentro de substituições marcadas, e aplicações de $\rightarrow_{\underline{x}}$ no resto do corpo do termo.

```
Inductive lab_x_i: pterm → pterm → Prop :=
| xi_from_bx_in_les: ∀ t1 t2 t2',
                          lab_term (t1 [[ t2 ]]) →
                          (sys_Bx t2 t2') →
                          lab_x_i (t1 [[ t2 ]]) (t1 [[ t2' ]])
| xi_from_x : ∀ t t',
```

lab_term t →
lab_sys_x t t' →
lab_x_i t t'.

Podemos agora formalizar a redução interna:

```
Notation "t ->[lx_i] u":= (ext_lab_EE_ctx_red lab_x_i t u) .
```

Assim, $\lambda\underline{ex}^i$ é formalizada como a redução "t -->[lx_i] u". Por outro lado, a relação $\lambda\underline{ex}^e$ formalizada como a redução "t -->[lx_e] u".

Como consequência dos diversos fechos e equações utilizadas, a manipulação dos construtores da redução se torna trabalhosa, pois precisamos lidar com quantificadores existenciais, conjunções, fechos transitivos, fechos contextuais, etc.

Uma estratégia utilizada para facilitar as provas foi abstrair, em lemas auxiliares, vários problemas destas que se repetem. Podemos então tratar tais problemas com um contexto limpo, facilitando as provas por indução.

Para cada fecho, definimos um lema para se realizar a redução em um termo maior, a partir da redução em um subtermo. Estes lemas são úteis para se evitar que tenhamos que destrinchar as reduções dentro da prova principal, facilitando muito o processo. Para cada construtor de termo, temos um lema para associar o construtor a um fecho, mais especificamente os fechos `lab_EE_ctx_red`, `ext_lab_EE_ctx_red` e também para o fecho transitivo-reflexivo de `lab_contextual_closure`.

A maioria destes lemas são simples de se resolver. Como de costume, os casos que lidam com abstrações e substituições precisam de um cuidado especial. Neles, precisamos assumir que valem os predicados `red_rename R` e `red_lab_regular' R` para a relação `R` sobre o qual estamos realizando o fecho. Isto porque iniciamos as provas escolhendo uma variável "nova", que não ocorre em nenhum dos termos do contexto. Isto auxilia a lidar com a quantificação necessária nas hipóteses de indução e construtores, em que precisamos de uma variável nova em um conjunto L. Precisamos então utilizar as propriedades de renomeamento e regularidade para poder substituir essa variável.

Além disso, alguns lemas para lidar com a relação entre as reduções e equações foram necessários:

```
Lemma EE_presv_ie: ∀ t t' u u', t =EE u → u' =EE t' → ((u ->[lx_i] u' ∨ u ->[lx_e]
u') → (t ->[lx_i] t' ∨ t ->[lx_e] t')).
```

```
Lemma EE_presv_lab_lex: ∀ t t' u u', t =EE u → u' =EE t' → ((u ->[lex] u') → (t
->[lex] t')).
```

Novamente, esses lemas evitam que precisemos adentrar na definição das equações, reduzindo o tamanho das provas principais, e são facilmente provados observando a transitividade da relação de equivalência `=EE`.

Podemos então partir para o resultado principal deste trabalho, ou seja, a prova do Teorema 2, formalizado como o teorema `lab_ex_eq_i_e`.

`Theorem` *lab_ex_eq_i_e*: $\forall\ t\ t'$, *lab_term* $t \to (t \text{ –>}[lex]\ t' \leftrightarrow (t \text{ –>}[lx\_i]\ t' \vee t \text{ –>}[lx\_e]\ t'))$.

Dividimos a prova em dois lemas, cada um representado uma direção da equivalência.

`Lemma` *lab_ex_impl_i_e*: $\forall\ t\ t'$, *lab_term* $t \to t \text{ –>}[lex]\ t' \to (t \text{ –>}[lx\_i]\ t' \vee t \text{ –>}[lx\_e]\ t')$.

Demonstração. Escolhemos, para cada redução possível feita pela relação `-->[lex]`, a redução apropriada entre a interna e a externa. Para isto, abrimos a definição da relação `-->[lex]` e fazemos indução no fecho contextual, ou seja, fazemos indução no predicado `lab_contextual_closure lab_sys_lx t t'`. O caso base é tratado no lema auxiliar `lab_sys_x_i_e`, que relaciona a relação `lab_sys_lx` com as relações interna e externa, e é feito com análise de casos simples nos construtores da relação `lab_sys_lx`.

Nos passos indutivos, utilizamos o lema `EE_presv_ie` para adequar o objetivo à hipótese de indução, substituindo os termos dados pelos termos equivalentes, obtidos pela definição da redução `-->[lex]`, podendo assim aplicar a hipótese. No caso em que a redução é feita dentro da substituição marcada, devemos obrigatoriamente realizar a redução interna. Em todos os outros casos, realizamos a prova tanto para a redução interna quanto para a externa. Nos casos em que lidamos com abstrações e substituições, são necessários os lemas de renomeamento mencionados na subseção 2.3.1. □

`Lemma` *lab_ie_impl_ex*: $\forall\ t\ t'$, *lab_term* $t \to (t \text{ -->}[lx\_i]\ t' \vee t \text{ -->}[lx\_e]\ t') \to t \text{ -->}[lex]\ t'$.

Demonstração. A prova deste lema é dividida em duas partes: quando a redução realizada é a interna, e quando é a externa.

No caso da interna, a indução é feita no fecho do predicado interno, ou seja, é feita no predicado `ext_lab_contextual_closure lab_x_i t t'`. O caso base é feito apenas analisando os construtores da relação `lab_x_i`, e casando com o construtor adequado de lab_sys_lx. Nos passos indutivos, fazemos de maneira análoga ao lema anterior: utilizamos agora o predicado `EE_presv_lab_lex` para ajustar o objetivo à hipótese, e utilizamos os lemas relacionando as reduções e

fechos aos construtores, mencionados acima, para reduzir o objetivo à redução dada como hipótese. Novamente, no caso de abstrações e substituições, precisamos dos lemas de renomeamento.

No caso da externa, o processo é exatamente o mesmo. A diferença é que no passo base, fazemos a análise de casos na redução `sys_Bx` e, além disso, não temos que lidar com o caso da redução dentro de substituição marcada. □

Com ambos os lemas completos, a prova do teorema `lab_ex_eq_i_e` se reduz a apenas aplicá-los, como visto a seguir.

```
Theorem lab_ex_eq_i_e: ∀ t t', lab_term t → (t –>[lex] t' ↔ (t –>[lx_i] t' ∨ t –>[lx_e] t')).
Proof.
    split.
    intros; apply lab_ex_impl_i_e; auto.
    intros; apply lab_ie_impl_ex; auto.
Qed.
```

Assim, terminamos a formalização da prova de que a redução do sistema com substituições marcadas, $\lambda\underline{ex}$, é equivalente à união das reduções interna e externa, $\lambda\underline{ex}^i$ e $\lambda\underline{ex}^e$.

Capítulo 3

Conclusão

Cálculos de substituição explícita são de interesse prático por servirem como um framework formal para o estudo de propriedades de sistemas reais, como implementações de linguagens funcionais e assistentes de prova (15). Desta forma, uma formalização se torna interessante pois fornece uma maneira mecânica de verificar estes sistemas. Estes também são importantes no estudo de propriedades do próprio cálculo λ (16; 17), pois algumas destas se tornam mais fáceis de serem provadas em um sistema com substituições explícitas, bastando assim mostrar a preservação da propriedade entre os cálculos.

Neste trabalho, foi continuada a formalização [1] do cálculo λex, iniciada em (27). Em um primeiro momento, foi dado enfoque na continuação da construção da teoria, em especial realizando provas de preservação de propriedades do cálculo pelas classes de equivalência. Após isto, definimos, dentro do sistema, todas as definições e propriedades relacionadas às substituições marcadas, e provamos suas características principais. Iniciamos, então, a prova da propriedade IE, realizando a prova formal da equivalência entre a redução do sistema estendida para substituições marcadas, $\lambda\underline{ex}$, e a união das reduções interna e externa, $\lambda\underline{ex}^i$ e $\lambda\underline{ex}^e$.

Como trabalho futuro, queremos estudar propriedades de normalização do sistema, procurando uma definição mais fácil de trabalhar para o conceito de **normalização forte**. Com isto, poderemos concluir a prova da propriedade IE dentro do sistema, finalizando, assim, a formalização do cálculo λex.

[1] A versão atual desta formalização está disponível em https://github.com/Lucas1993/LambdaEX_TCC

Referências

[1] Appel, K. e W. Haken: *Every planar map is four colorable. part i: Discharging*. Illinois J. Math., 21(3):429–490, setembro 1977. http://projecteuclid.org/euclid.ijm/1256049011.

[2] Gonthier, G.: *A computer-checked proof of the four colour theorem*. Relatório Técnico, Microsoft Research Cambridge, 2008.

[3] Gonthier, G.: *The Four Colour Theorem: Engineering of a Formal Proof*, capítulo Computer Mathematics, páginas 333–333. Computer Mathematics. Springer Science + Business Media, 2008. http://dx.doi.org/10.1007/978-3-540-87827-8_28.

[4] Wiedijk, Freek: *Comparing Mathematical Provers*, páginas 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, ISBN 978-3-540-36469-6. http://dx.doi.org/10.1007/3-540-36469-2_15.

[5] Geuvers, H.: *Proof assistants: History, ideas and future*. Sadhana, 34:3–25, 2009.

[6] Coquand, Thierry e Gerard Huet: *The calculus of constructions*. Inf. Comput., 76(2-3):95–120, fevereiro 1988, ISSN 0890-5401. http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[7] *A short introduction to Coq*. https://coq.inria.fr/a-short-introduction-to-coq. Accessed: 2016-05-09.

[8] Nahas, M.: *A tutorial by mike nahas*. https://coq.inria.fr/tutorial-nahas. Accessed: 2016-05-09.

[9] Pierce, B. C.: *Software Foundations*. http://www.cis.upenn.edu/~bcpierce/sf/. Accessed: 2016-05-09.

[10] Church, A.: *An unsolvable problem of elementary number theory*. American Journal of Mathematics, 58(2):345–363, 1936.

[11] Bruijn, N. G. De: *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem*. INDAG. MATH, 34:381–392, 1972.

[12] Ayala-Rincón, M. e C. Muñoz: *Explicit Substitutions and All That*. Revista Colombiana de Computación, 1(1):47–71, 2000.

[13] Charguéraud, A.: *The locally nameless representation.* Journal of Automated Reasoning, 49(3):363–408, 2012, ISSN 1573-0670. http://dx.doi.org/10.1007/s10817-011-9225-2.

[14] Barendregt, H. P.: *The Lambda Calculus, its syntax and semantics.* North Holland, 1985.

[15] Lévy, Jean Jacques e Luc Maranget: *Explicit Substitutions and Programming Languages*, páginas 181–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, ISBN 978-3-540-46691-8. http://dx.doi.org/10.1007/3-540-46691-6_14.

[16] Accattoli, Beniamino e Luca Paolini: *Call-by-value solvability, revisited.* Em *Proceedings of the 11th International Conference on Functional and Logic Programming*, FLOPS'12, páginas 4–16, Berlin, Heidelberg, 2012. Springer-Verlag, ISBN 978-3-642-29821-9. http://dx.doi.org/10.1007/978-3-642-29822-6_4.

[17] Accattoli, Beniamino e Ugo Dal Lago: *Beta reduction is invariant, indeed.* Em *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, páginas 8:1–8:10, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2886-9. http://doi.acm.org/10.1145/2603088.2603105.

[18] Lins, R. D.: *A new formula for the execution of categorical combinators*, páginas 89–98. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986, ISBN 978-3-540-39861-5. http://dx.doi.org/10.1007/3-540-16780-3_82.

[19] Rose, Kristoffer Høgsbro: *Explicit cyclic substitutions.* Em *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems*, CTRS '92, páginas 36–50, London, UK, UK, 1993. Springer-Verlag, ISBN 3-540-56393-8. http://dl.acm.org/citation.cfm?id=648339.756055.

[20] Bloo, Roel e Kristoffer H. Rose: *Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection.* Em *IN CSN-95: COMPUTER SCIENCE IN THE NETHERLANDS*, páginas 62–72, 1995.

[21] Hardin, Thèrése e Jean Jacques Lévy: *a Confluent calculus of substitutions.* France-Japan Artificial Intelligence and Computer Science Symposium, 1989.

[22] David, René e Bruno Guillaume: *A λ-calculus with explicit weakening and explicit substitution.* (11):169–206, 2001.

[23] Kesner, Delia e Stéphane Lengrand: *Resource operators for λ-calculus.* Inf. Comput., 205(4):419–473, abril 2007, ISSN 0890-5401. http://dx.doi.org/10.1016/j.ic.2006.08.008.

[24] Melliès, P. A.: *Typed λ-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95.* 902, 1995.

[25] Guillaume, B.: *The λs_e-calculus Does Not Preserve Strong Normalization.* J. of Func. Programming, 10(4):321–325, 2000.

[26] Kesner, D.: *A Theory of Explicit Substitutions with Safe and Full Composition*, volume 5, páginas 1–29. 2009.

[27] Carvalho Segundo, Washington de, Flávio L. C. de Moura e Daniel Ventura: *Formalizing a named explicit substitutions calculus in coq*. Em *Joint Proceedings of the MathUI, OpenMath and ThEdu Workshops and Work in Progress track at CICM co-located with Conferences on Intelligent Computer Mathematics (CICM 2014), Coimbra, Portugal, July 7-11, 2014.*, volume 1186, 2014. http://ceur-ws.org/Vol-1186/paper-19.pdf.