Formalizando uma prova da confluência do cálculo lambda em Coq

Gabriel Alves Castro

1 Introdução

O cálculo lambda é um formalismo importante que fundamenta o paradigma de programação funcional. Neste paradigma, um programa é uma função que, por sua vez pode ser composta para construir outros programas. A principal característica que diferencia o paradigma funcional do paradigma imperativo de programação é o fato de que a ordem da definição das funções é irrelevante, o que permite que o processo de avaliação de um programa ocorra de diferentes formas, i.e. em diferentes ordens. No entanto, estas diferentes ordens resultam sempre na mesma resposta, e esta propriedade é conhecida como confluência. Existem diversas provas de confluência para o cálculo lambda, e uma das mais populares é uma prova feita por H. Barendregt [1]. Neste trabalho, apresentamos os primeiros resultados de um projeto que visa formalizar a prova de confluência do cálculo lambda sem tipos de Barendregt no assistente de provas Coq.

A utilização de ferramentas formais, como assistentes de prova, para a construção de programas corretos é cada vez mais frequente. A preocupação em construir programas corretos, antes restrita basicamente a sistemas críticos como aqueles relacionados à equipamentos médicos, programas utilizados no sistema financeiro ou em aeronaves, hoje se aplica a praticamente qualquer programa de computador. Neste contexto, os assistentes de prova têm conquistado cada vez mais espaço porque são capazes de fornecer garantias absolutas obtidas por meio de provas matemáticas.

O cálculo lambda constitui um importante modelo teórico de computação, baseado no conceito de função, e sobre o qual se constrói todo o paradigma funcional de programação. Atualmente diversos sistemas computacionais, que vão desde linguagens de programação, como Haskell e as linguagens da família ML, a assistentes de prova como Coq, PVS e Isabelle/HOL, são baseados neste modelo.

Este projeto possui duas linhas fundamentais de pesquisa: Inicialmente, estudamos propriedades de extensões do cálculo lambda com substituições explícitas, que podem ser vistos como formalismos intermediários entre o cálculo lambda e suas implementações. Estes formalismos intermediários são utilizados tanto como metalinguagem para estudo do próprio cálculo lambda, como para estudos de suas implementações. Em particular, estudamos a propriedade da confluência

que caracteriza o determinismo do processo computacional.

1.1 O ambiente LNR

A notação de nomes locais (LNR) consiste em um ambiente desenvolvido em Coq por A. Charguéraud [2], onde as variáveis são divididas em duas classes: índices de DeBruijn que são utilizados para codificar as variáveis ligadas, e nomes (como usual para representar as variáveis livres). Dessa forma podemos ter um conjunto de símbolos nesta notação que não sejam válidos semanticamente. Vamos exemplificar:

Dada a abstração $\lambda.0$ sabemos que o índice 0 representa a variável ligada à abstração que apresentamos. Estando esse índice portanto ligado diretamente à abstração apresentada, e sendo dessa maneira válido semanticamente. Porém, sintaticamente o índice pode ser um valor iteiro k qualquer. Assim, por exemplo, ao fazermos $\lambda.1$, estamos construindo uma expressão que não é válida pois o índice não está ligado a nenhum abstrator. Queremos representar variáveis ligadas por meio de índices, e neste caso o index 1 não está ligado à nenhuma abstração, portanto não representa uma variável ligada e não é válido para nossa representação.

Assim podemos definir o conceito de pré-termo como sendo a gramática contendo os seguintes construtores:

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

O construtor *pterm_bvar* é utilizado para representar as variáveis ligadas, *pterm_fvar* para variáveis livres, *pterm_app* é o construtor de uma aplicação, *pterm_abs* é o construtor de abstrações e *pterm_labs* é o construtor de abstrações marcadas.

Para trabalhar apenas com termos válidos, i.e. expressões sem índices de DeBruijn que não estejam ligados a nenhuma abstração ou substituição, precisamos definir algumas operações que permitirão à máquina fazer as devidas buscas e comparações. Estas operações serão definidas mais abaixo. A definição da operação "variable opening", ou seja, abertura de variáveis é dada abaixo:

```
Fixpoint open_rec (k : nat) (u : pterm) (t : pterm) : pterm :=
  match t with
  | pterm_bvar i ⇒ if k === i then u else (pterm_bvar i)
  | pterm_fvar x ⇒ pterm_fvar x
  | pterm_app t1 t2 ⇒ pterm_app (open_rec k u t1) (open_rec k u t2)
  | pterm_abs t1 ⇒ pterm_abs (open_rec (S k) u t1)
  | pterm_labs t1 ⇒ pterm_labs (open_rec (S k) u t1)
  end.
```

Com isso, a operação de abertura recursiva apenas para index's 0 é definida especialmente como "open", onde u é o nome de uma variável qualquer e t é um pré-termo, logo abaixo:

```
Definition open t u := open_rec 0 u t.
```

Esta operação é responsável por substituir todos os índices "k", por uma variável com nome qualquer. Por exemplo, digamos que tenhamos o pré-termo $\lambda.0y$, assim, ao aplicar a operação $0 > x\lambda.0y$ teremos o seguinte resultado: $\lambda.xy$.

Esta definição será extremamente útil nas provas mais abaixo, devido à maior facilidade com relação à trabalhar com qualquer index k, e para garantir que estamos trabalhando com termos válidos.

De qualquer forma, a operação como explicada mais acima, onde k é o index de De Bruijin, u o nome de uma variável qualquer e t o pré-termo que será aberto recursivamente no index k é definida como se segue:

```
Notation "{ k ~¿ u } t":= (open_rec k u t) (at level 67).
```

Notação para representar a abertura de um termo, substituindo as variáveis ligadas por qualquer tipo de pré-termo.

```
Notation "t ^^ u":= (open t u) (at level 67).
```

Notação para representar abertura de um pré-termo utilizando uma variável livre x:

```
Notation "t ^ x":= (open t (pterm_fvar x)).
```

A definição de termo é dada logo abaixo:

```
Inductive term : pterm → Prop :=
  | term_var : ∀ x,
      term (pterm_fvar x)
  | term_app : ∀ t1 t2,
      term t1 →
      term t2 →
      term (pterm_app t1 t2)
  | term_abs : ∀ L t1,
      (∀ x, x "notin L → term (t1 ^ x)) →
      term (pterm_abs t1).
```

Um termo é válido na notação de nomes locais, quando este termo é um termo fechado. Para ser um termo fechado, seguimos uma definição recursiva, onde toda variável livre, e portanto nomeada é fechada, toda aplicação é fechada, se seus dois termos internos são fechados, e toda abstração é fechada, se todos os termos que fazem parte da mesma também são fechados. Veja que esta é exatamente a definição que temos logo acima. Podemos deixar o entendimento mais fácil com a seguinte definição: Um termo é um pré-termo sem nenhuma variável ligada inválida, ou seja, nenhum index não ligado à uma abstração de alguma maneira.

Dessa forma também é interessante definir o conceito de corpo, como sendo todo pré-termo que após uma abertura no index 0, por uma variável nomeada livre x, torna-se um termo fechado. A definição está logo abaixo:

```
Definition body t := ∃ L, ∀ x, x “notin L → term (t ^ x).
```

Perceba que a definição de body foi utilizada na definição recursiva de termo, para representar as abtrações válidas semanticamente, segundo os conceitos aqui já apresentados.

Para realizar a prova da confluência é necessária a definição de um termo marcado. Um termo marcado contém exatamente as mesmas propriedades de um termo, exceto que, pode possuir abstrações marcadas. Abstrações marcadas são aquelas que estão postas em uma aplicação válida, aonde pode ser aplicada uma B-redução. Abstrações que não fazem parte de uma aplicação, não podem ser abstrações marcadas. Podemos ver essa definição em termos recursivos logo abaixo:

```
Inductive lterm : pterm → Prop :=
  | lterm_var : ∀ x,
      lterm (pterm_fvar x)
  | lterm_app : ∀ t1 t2,
      lterm t1 →
      lterm t2 →
      lterm (pterm_app t1 t2)
  | lterm_abs : ∀ L t1,
      (∀ x, x “notin L → lterm (t1 ^ x)) →
      lterm (pterm_abs t1)
  | lterm_labs : ∀ L t1 t2,
      (∀ x, x “notin L → lterm (t1 ^ x)) →
      lterm t2 →
      lterm (pterm_app (pterm_labs t1) t2).
```

Assim, também é possível definir o corpo marcado, como sendo o pré-termo que possui dentro de si uma abstração marcada, aonde após uma abertura recursiva do index 0 com uma variável nomeada livre qualquer x, torna-se um termo marcado:

```
Definition lbody t := ∃ L, ∀ x, x “notin L → lterm (t ^ x).
```

2 Definições técnicas e explanação teórica

Para auxiliar nas provas são necessárias algumas definições no assistente de provas. Existem casos em que é melhor seguir uma prova por indução na estrutura, nestes casos, o assistente consegue lidar bem com seus próprios comandos. No entanto, quando precisamos trabalhar com provas no tamanho dos termos, muitas vezes são necessárias definições para contar o tamanho do tipo que estamos lidando. E posteriormente, essas definições são utilizadas para definir induções no tamanho específicas para o domínio em que estamos trabalhando. Não vamos apresentar estas definições neste trabalho, por serem muito técnicas e poderem ser facilmente encontradas em outros trabalhos da literatura como em [3] que aborda a construção de provas indutivas usando o Coq.

No trabalho de [2] e na notação de nomes locais é necessária a definição de operações de abertura e fechamento dos termos. As operações de abertura e fechamento manipulam os index's e nomes, e têm como objetivo controlar e tornar possível a decisão de se um termo é ou não fechado, ou seja, possui uma sintaxe válida. Demonstramos como as operações de abertura (open) são definidas, mas não iremos mostrar essas definições para as operações de fechamento neste trabalho, pois as mesmas podem ser encontradas no próprio trabalho de [2], e não foram utilizadas em nossa provas.

Existe outra maneira de saber se um pré-termo é localmente fechado. Além do uso do open, podemos definir um tipo recursivo chamado "lc_at". A ideia é trabalhar com os index's de modo a verificar se todos possuem ligação com uma abstrção, e caso sim, o termo será um termo fechado. Ser um termo fechado, como já dito, significa ser um termo válido para o λ-cálculo. A comparação realizada para os index's refere-se a se estes são menores do que um valor k, ligado ao número de abstrações.

Para um termo ser realmente localmente fechado segundo essa definição, ele precisa ser lc_at 0, ou seja, ele não pode conter nenhum index sem ligações válidas com abstrações. Entendemos melhor essa propriedade ao analisar a definição recursiva do lc_at. Toda variável livre é localmente fechada. A medida que um termo será localmente fechado para uma aplicação, se seus dois subtermos também foram localmente fechados. E para uma abstração, o será se o termo envolvido pela abstração for localmente fechado para k + 1. Para a variável ligada, portanto, o termo será localmente fechado caso o index i que representa a variável ligada seja menor do que k. Veja a definição abaixo:

```
Fixpoint lc_at (k:nat) (t:pterm) : Prop :=
  match t with
  | pterm_bvar i ⇒ i ¡ k
  | pterm_fvar x ⇒ True
  | pterm_app t1 t2 ⇒ lc_at k t1 ∧ lc_at k t2
  | pterm_abs t1 ⇒ lc_at (S k) t1
  | pterm_labs t1 ⇒ lc_at (S k) t1
  end.
```

O lc_at pode ser usado em diversos teoremas matemáticos relacionados tanto ao open, quanto à definição de termos. Esses teoremas podem facilitar provas com nível de abstração mais elevado na teoria do cálculo lambda, permitindo a equivalência entre diferentes tipos e propriedades.

3 Propriedades importantes do cálculo lambda e suas provas

Precisamos definir e provar diversas propriedades do cálculo lambda que irão ser muito úteis para a construção das provas de confluência que estamos construindo. Nesse sentido, muitas das propriedades que poderiam parecer simples

de provar, no assistente de provas essa tarefa torna-se muitas vezes muito mais complicada.

Uma das propriedades mais interessantes da notação sem nomes locais que empregamos é que ela não necessita da alfa conversão, facilitando e tornando decidíveis para a máquina diversas operações que antes não o seriam. Podemos provar essa propriedade por meio da definição de lc_at, como se segue no lema abaixo, onde sua prova no coq segue de maneira trivial. Neste lema mostramos que se dois nomes são utilizados como variáveis livres em uma estrutura análoga, então eles são localmente fechados em um nível m.

`Lemma` *lc_at_open_rec_rename*: $\forall\ t\ x\ y\ m\ n$, *lc_at* m (*open_rec* n (*pterm_fvar* x) t) $\rightarrow$ *lc_at* m (*open_rec* n (*pterm_fvar* y) t).

Outra propriedade importante é o fato de que se um pré-termo é localmente fechado no nível n, então ele também é localmente fechado para qualquer nível maior do que n. Esta propriedade pode ser provada segundo a definição do lema abaixo, e sua prova no coq é trivial.

`Lemma` *lc_at_weaken*: $\forall\ t\ n\ m$, $n \leq m \rightarrow$ *lc_at* $n\ t \rightarrow$ *lc_at* $m\ t$.

É importante demonstrar também a relação de sucessor com relação à termos localmente fechados e abertos com open. Nesse caso, se um pré-termo é localmente fechado em m e é aberto em m com a variável livre qualquer x, então ele continuará localmente fechado para m + 1. A prova desta propriedade é um pouco mais complexa, mas pode ser realizada com um pouco de esforço no coq.

`Lemma` *lc_at_open*: $\forall\ t\ m\ x$, *lc_at* m ({m ~¿ *pterm_fvar* x} t) $\leftrightarrow$ *lc_at* ($S\ m$) t.

Assim, também somos capazes de provar uma das propriedades que já havíamos citado acima, a de que um termo é um pré-termo fechado para o index 0. Essa conversão pode permitir trabalhar a definição de lc_at ao invés de term, quando essa for mais conveniente. A prova para esse teorema é simples, mas necessita da prova do lema lc_at_open para ser concluída.

`Lemma` *term_to_lc_at* : $\forall\ t$, *term* $t \rightarrow$ *lc_at* $0\ t$.

O mesmo segue para o termo marcado, por definição.

`Lemma` *lterm_to_lc_at* : $\forall\ t$, *lterm* $t \rightarrow$ *lc_at* $0\ t$.

Muitas vezes pode ser importante considerar se um pré-termo possui um index livre. Isto para checar por exemplo, se há a possibilidade de uma redução de uma abstração. Isso pode ser realizado por meio da definição abaixo:

```
Fixpoint has_free_index (k:nat) (t:pterm) : Prop :=
  match t with
    | pterm_bvar n ⇒ if (k === n) then True else False
    | pterm_fvar x ⇒ False
    | pterm_app t1 t2 ⇒ (has_free_index k t1) ∨ (has_free_index k t2)
    | pterm_abs t1 ⇒ has_free_index (S k) t1
    | pterm_labs t1 ⇒ has_free_index (S k) t1
```

`end`.

Respeitando a questão da alfa conversão, podemos demonstrar que se um pre-termo recebe uma substituição pela variável livre x, então ao receber de uma outra variável qualquer y uma substituição, este pré-termo continuaria sendo um termo.

`Lemma` *term_rename*: $\forall\ t\ x\ y$, *term* $(t\ \hat{}\ x) \rightarrow$ *term* $(t\ \hat{}\ y)$.

também precisamos provar que, para um termo qualquer, qualquer substituição não tem efeito. Essa propriedade pode ser muito valiosa para algumas provas, principalmente a prova do lema da substituição, e é definida como no lema abaixo:

`Lemma` *subst_term*: $\forall\ t\ u\ n$, *term* $t \rightarrow \{n\ \tilde{}¿\ u\}\ t = t$.

Por definição o mesmo também vale para um termo marcado:

`Lemma` *subst_lterm*: $\forall\ t\ u\ n$, *lterm* $t \rightarrow \{n\ \tilde{}¿\ u\}\ t = t$.

O lema da substituição é muito importante quando estamos realizando algumas operações e provas no cálculo lambda. Abaixo temos a sua definição como tentamos o provar, no entanto, ainda sem sucesso. O lema da substituição afirma que, realizar uma substituição A após já ter realizado uma substituição B prévia, é equivalente à realizar a substituição posterior A primeiro, e a substituição posterior B depois, mas é necessário realizar uma substituição dentro do termo que será a substituição na substituição A pela mesma substituição B. Vamos exemplificar na notação que estamos utilizando: $n > uj > at = j > an > j > aut$. Veja que algumas propriedades devem ser levadas em consideração: i e j devem ser index's distintos, para não haver dessa forma inconformidades. Sugerimos a realização de provas em estrutura de árvores de prova (citação?), para que essa situação seja melhor compreendida. A definição do lema, para a construção de sua prova em assistente de provas está logo abaixo:

`Lemma` *subst_lemma_lterms*: $\forall$ (*t1 t2 t3*: *pterm*) (*i j*:*nat*), *lterm t2* $\rightarrow$ *lterm t3* $\rightarrow i \neq j \rightarrow \{j\ \tilde{}¿\ t3\}\ (\{i\ \tilde{}¿\ t2\}\ t1) = \{i\ \tilde{}¿\ t2\}\ (\{j\ \tilde{}¿\ t3\}\ t1)$.

4 Definição de beta redução

A operação da beta redução é muito importante para o cálculo lambda. Uma beta redução consiste na operação de reduzir uma aplicação com uma abstração à esquerda, à substituição da abstração em todos os lugares em que o index local ocorrer no termo envolvido pela abstração em questão. No assistente de provas, para definir essa operação, precisamos seguir alguns passos, devido à quantidade de possíveis eventos quando da aplicação da operação. Estes passos se encontram logo abaixo.

Inicialmente criamos a propriedade de relação, advinda da teoria dos conjuntos. Uma relação é um binômio (a,b), onde a e b pertencem à um mesmo conjunto de tipos, como por exemplo, o conjunto dos números reais. No assistente de provas isso pode ser feito como abaixo: Se um determinado tipo A, implica em um determinado tipo A, então temos uma propriedade.

```
Definition Rel (A:Type) := A → A → Prop.
```

4.1 Definição de redex

Para trabalhar com a beta redução é muito importante trabalhar com o conceito de redex. Um redex nada mais é do que uma aplicação que possui uma abstração marcada ou não à sua esquerda. Dessa forma precisamos definir o feixo contextual transitivo, ou seja, precisamos ajudar a máquina a entender quando existem ou não redex's em um termo. Um feixo contextutal transitivo define uma relação direta entre dois termos, estes termos seguem algumas restrições. Podemos dizer que, estes termos são o termo anterior à uma beta redução, e o termo posterior à aplicação de uma beta redução em um redex específico de um termo e estes dois termos participam portanto de uma relação. Como a definição de termo é recursiva, e uma aplicação ocorre entre dois termos, então precisamos deixar claro para a máquina que, primeiro as beta reduções que ocorrerem tanto à esquerda, quanto à direita, em uma aplicação relacionam o termo anterior, ao termo resultante. Além disso, também precisamos relacionar os termos resultantes de uma redução em uma abstração, estão relacionados com os termos que possuiam a abstração.

Assim podemos definir as operações de beta redução, e da enésima beta redução. rule_b se refere à definição de uma relação onde um redex é substituiído pela termo resultante da substituição de de todos os index's ligados de um termo qualquer t, por uma variável u. Na prática, esta é a definição de beta redução, quando esta é posta em conjunto com o feixo contextual transitivo, como visto na definição do símbolo de beta redução para a máquina: "–¿B". A operação da enésima beta redução é feita por meio da aplicação sucessiva e recursiva da operação da beta redução, por meio da regra refltrans, explicada mais acima.

```
Inductive rule_b : Rel pterm :=
  reg_rule_b : ∀ (t u:pterm), body t → term u →
    rule_b (pterm_app(pterm_abs t) u) (t ^^ u).
Notation "t –¿B u":= (contextual_closure rule_b t u) (at level 60).
Notation "t –¿¿B u":= (refltrans (contextual_closure rule_b) t u) (at level
60).
```

O mesmo deve ser definido para o conjunto dos termos marcados. Neste caso, precisamos tomar o cuidado de diferenciar as abstrações das abstrações marcadas, e tratar cada um dos casos, apesar de sabermos serem análogos, a máquina não o sabe.

```
Inductive rule_lb : Rel pterm :=
  | reg_rule_bb : ∀ (t u:pterm), body t → term u →
```

```
    rule_lb (pterm_app(pterm_abs t) u) (t ^^ u)
  | reg_rule_lb : ∀ (t u:pterm),
    body t → term u →
    rule_lb (pterm_app(pterm_labs t) u) (t ^^ u).
Notation "t –¿lB u":= (lcontextual_closure rule_lb t u) (at level 60).
Notation "t –¿¿lB u":= (refltrans (lcontextual_closure rule_lb) t u) (at level
60).
```

5 Prova da confluência

5.1 O que é confluência

A confluência no cálculo lambda se define como: Dados dois termos iniciais provenientes de um único termo, a partir de qualquer conjunto de reduções beta (duas bifurcações quaisquer na árvore de reduções) é sempre possível chegar ao mesmo termo resultante (considerando a alfa equivalência). Assim, esperamos provar que o cálculo lambda é confluente. Tal característica (confluência) permite dizer que o cálculo lambda é determinístico, sendo possível sempre dadas condições iniciais prever um determinado resultado, ao operar as operações corretas (neste caso, reduções).

5.2 Operações necessárias para a prova da confluência

Já definimos os termos marcados, por meio da criação de uma "marca" em determinados redex's. Um redex é toda abstração que é base para uma determinada aplicação definida em um determinado termo. Dessa forma, é necessário definir algumas operações que possam trabalhar com os termos marcados, permitindo que estas marcas sejam tanto colocadas, quanto retiradas. Geralmente as marcas são criadas na própria definição de uma determinada prova para então, serem retiradas depois, comparando um termo marcado à um termo não marcado, após uma determinada sequência de operações do λ-cálculo, permitindo assim provar determinada hipótese, pois com a marca é possível acompanhar a o estado de um determinado redex após sucessivas operações.

Para isso, uma das operações definidas é a operação de apagamento (erase) de marcas, ou melhor, a operação de apagar. A operação de apagar consiste em, quando da sua aplicação em um termo, apagar todas as suas (se houverem) marcas, mas preservando a sua estrutura (sem reduzir os redex's). Essa operação foi definida recursivamente como se segue abaixo, sendo propagada para dentro de cada termo, com atenção especial à abstração marcada, que transforma-se em abstração, e mantém a propagação do erase. Assim, após a aplicação da operação, um termo marcado se tornará um termo sem marcas, e um termo sem marcas não sofrerá nenhuma alteração.

```
Fixpoint erase (t:pterm) : pterm :=
  match t with
  | pterm_app t1 t2 ⇒ pterm_app (erase t1) (erase t2)
```

```
  | pterm_abs t1 ⇒ pterm_abs (erase t1)
  | pterm_labs t1 ⇒ pterm_abs (erase t1)
  | _ ⇒ t
  end.
```

Dessa forma, também é necessária a definição da função phi. A função phi também trabalha com marcas, assim como o erase, no entanto possui um funcionamento um pouco distinto. A função phi possui o papel de, ao ser aplicada em um termo, reduzir todos os redex's marcados (abstrações com marcas com um determinado termo sendo aplicado). Dessa forma, após a aplicação da operação phi um termo marcado torna-se um termo sem marcas, porém difere estruturalmente do seu estado anterior, pelo fato de agora possuir os antigos redexs marcados reduzidos. Perceba que um termo sem marcas, ao receber a aplicação de phi permanece inalterado, por definição. Definimos a operação como se segue logo abaixo, recursivamente, a operação phi é propaga pelo termo que sofre a aplicação, reduzindo os redex's marcados, o encontrar uma aplicação onde o termo mais à esquerda é uma abstração marcada.

```
Fixpoint phi (t:pterm) : pterm :=
  match t with
  | pterm_app t1 t2 ⇒ match t1 with
                      | pterm_labs t1' ⇒ (phi t1') ^^ (phi t2)
                      | _ ⇒ pterm_app (phi t1) (phi t2)
                      end
  | pterm_abs t1 ⇒ pterm_abs (phi t1)
  | pterm_labs t1 ⇒ pterm_labs (phi t1)
  | _ ⇒ t
  end.
```

Com isso, podemos provar, por exemplo, que um termo marcado torna-se um termo após a aplicação da operação phi. A definição deste lema está como abaixo, e não é tão complexo realizar a sua prova no coq.

```
Lemma phi_term: ∀ t, lterm t → term (phi t).
```

Na prova de barendregt para a confluência do cálculo lambda também é muito importante provar a propriedade de que, se um termo M reduz para um termo N, então o resultado da aplicação da função phi em M, também reduz para o resultado da aplicação da função phi no termo N.

Também possuímos outra propriedade, que será muito importante para a prova da confluência do cálculo lambda. Esta se refere ao fato de que se M reduz para N em n passos, sendo M e N termos pertencentes ao conjunto dos termos marcados, então o resultado de aplicar a função de apagar marcas (erase) em M reduz ao resultado da aplicação da operação de apagar marcas em N. A definição para este lema está logo abaixo em duas possíveis formas. Até o momento, não conseguimos completar nenhuma das duas provas ainda no coq.

6 O strip_lemma

O strip_lemma é muito importante para a prova da confluência do cálculo lambda, utilizando a abordagem de Barendregt (citar). O strip_lemma prova a propriedade de que: Se um termo reduz em um passo para t1, e também reduz em n passos para t2, então existe um t3 tal que t1 reduz em n passos para t3 e t2 reduz em n passos para t3.

Perceba que essa propriedade ainda não prova a nossa definição de confluência, mas está um passo atrá de tal prova. Ao provar o strip_lemma e algumas outras propriedades, a prova da confluência do cálculo lamba é direta. De outra forma, podemos dizer que a prova da confluência no cálculo lambda é uma generalização do strip_lemma. Aida não conseguimos fechar a prova formal do strip_lemma, mas iremos apresentar logo abaixo os nossos avanços e apresentar quais os próximo passos que devemos seguir.

6.1 formalização da prova do strip_lemma

Para realizar a prova do strip_lemma iremos considerar os termos t, t1 e t2. Porém, para isso, iremos considerar que o termo t possui em sua composição o seguinte redex: pterm_app (pterm_labs Q) P, e apenas iremos levar em conta este redex marcado para construir as nossas operações. Ou seja, o termo deverá conter apenas um redex marcado, para podermos acompanhar o seu estado ao longo das operações de prova que serão realizadas. Dessa forma devemos considerar o seguinte: t1 é o termo obtido ao aplicar uma beta redução marcada em t, e reduzir especificamente o redex marcado apresentado. Nesse caso, portanto, aplicando phi(t) temos t1. Ou seja:

$phi(t) = t1$

Com isso, podemos dizer que existe um termo t' cuja erase(t) = t', e t' -¿B t1. Assim podemos reduzir t' para t2 via n reduções beta, e provar que t1 tabém reduz para t2 via n beta reduções. O que está sendo realizado neste caso é a manutenção de um redex marcado, após n beta reduções quaisquer, que não ocorrem no redex marcado. Em seguida o erase e a função phi são utilizados para demonstrar que os termos convergem em t2. Como esses termos são quaisquer e a única característica com a qual estamos trabalhando é a marca, então a prova pode ser generalizada para qualquer caso.

Exemplificando um pouco melhor, uma espécie de contador, como o abaixo poderia ser utilizado para se certificar de que o termo que está sendo trabalhado é um termo com apenas um redex marcado. Este é um dos caminhos de prova com o qual estamos trabalhando no projeto.

```
Fixpoint lredex_count (t:pterm):(nat) :=
match t with
  | pterm_bvar i ⇒ 0
  | pterm_fvar x ⇒ 0
  | pterm_app t1 t2 ⇒ (lredex_count t1) + (lredex_count t2)
  | pterm_abs t1 ⇒ (lredex_count t1)
  | pterm_labs t1 ⇒ 1 + (lredex_count t1)
```

```
end.
```

A prova será realizada por meio da indução na estrutura da beta redução de t para t1. Assim chegamos a quatro casos. Estes casos são relativos à própria estrutura da beta redução de t para t1. Ao provar os passos anteriores, a prova se completa por indução para o caso geral. No entanto, a definição atual do strip_lemma está muito generalizada, não havendo, por exemplo, um termo marcado definido com apenas uma marca que possa ser acompanhada durante a prova. Nesse caso, vamos utilizar a notação de nomes locais sem marcas e o trabalho de "cite{chatgerout} (citar certo) para nos ajudar a completar a prova mesmo de maneira generalizada, mas também apresentamos opções ilustrativas mais abaixo.

Dado esse contexto, dois caminhos podem ser seguidos: No primeiro, a definição do teorema poderia ser refeita, de uma maneira menos generalizada, abarcando as marcas e utilizando alguma ideia semelhante à apresentada na função lredex_count. O segundo caminho, seria a construção de diversos lemas auxiliares que permitiriam com que o teorema geral pudesse ser provado. Na realidade o sucesso em realizar esta prova provavelmente depende um caminho intermediário entre essas duas opções.

7 Conclusão

O presente trabalho desenvolveu parte do formalismo necessário para a construção de uma formalização de uma prova de confluência do cálculo lambda que utiliza duas classes de variáveis: índices de DeBruijn para variáveis ligadas, e nomes para variáveis livres. Esta formalização está sendo construída no assistente de provas Coq, uma ferramenta de código aberto e baseada em uma lógica de ordem superior conhecida como cálculo de construções indutivas.

Referências

[1] H. P. Barendregt. λ-calculi with types. *Handbook of Logic in Computer Science*, II, 1992.

[2] A. Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, pages 1–46, 2011.

[3] C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.