Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Brasília

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

Uma Formalização da Composicionalidade do Cálculo λ_{ex} em Coq

Flávio José Ferro Barros

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Flávio Leonardo Cavalcanti de Moura

Brasília
2010

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Mestrado em Informática

Coordenador: Prof. Dr. Mauricio Ayala Rincón

Banca examinadora composta por:

Prof. Dr. Flávio Leonardo Cavalcanti de Moura (Orientador) — CIC/UnB
Prof. Dr. Mauricio Ayala Rincón — CIC/UnB
Prof. Dr. Paulo Henrique de Azevedo Rodrigues — MAT/UFG

CIP — Catalogação Internacional na Publicação

Barros, Flávio José Ferro.

Uma Formalização da Composicionalidade do Cálculo λ_{ex} em Coq / Flávio José Ferro Barros. Brasília : UnB, 2010.
68 p. : il. ; 29,5 cm.

Tese (Mestrado) — Universidade de Brasília, Brasília, 2010.

1. Cálculo λ_{ex}, 2. cálculos de substituições explícitas,
3. Composicionalidade, 4. Confluência

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Dedicatória

Aos meus pais,

José Ribamar Souza Barros e Idália Selma Ferro Barros.

Agradecimentos

Este trabalho se tornou possível devido ao apoio de Deus e de diversas pessoas dentre as quais gostaria de destacar:

O professor Flávio Leonardo Cavalcanti de Moura pela paciência, empenho, sabedoria, compreensão, amizade e, principalmente, por ter me iniciado no campo da pesquisa. Gostaria de ratificar a sua competência, participação com diversas reuniões de estudos, correções, revisões na elaboração de artigos, sugestões que fizeram com que concluíssemos este trabalho. Todo este aprendizado será de fundamental importância para futura conduta profissional.

À meus pais e minha avó Maria de Lourdes Pereira Ferro pelo apoio sempre incondicional e pelos ensinamentos de vida.

Aos colegas, professores e funcionários do Departamento de Ciência da Computação da UnB.

Resumo

Apresenta-se uma formalização das propriedades de composicionalidade do Cálculo λ_{ex} em *Coq*. A abordagem utilizada baseia-se na lógica nominal de acordo com o trabalho desenvolvido por [3]. Mais especificamente estendemos a formalização do λ-cálculo contida neste trabalho de forma a incluir a operação de substituição explícita do cálculo λ_{ex}. Nessa abordagem, a α-equivalência coincide com a igualdade pré-construída de *Coq*, e os princípios de recursão e indução sobre classes de λ-termos possuem tratamento específico. Escolhemos trabalhar com o cálculo λ_{ex} por ser atualmente o único cálculo que satisfaz simultaneamente todas as propriedades desejáveis para um cálculo de substituições explícitas. Ele é uma extensão do λ_x com uma regra de reescrita para composição de substituições dependentes e uma equação para comutação de substituições independentes. O cálculo λ_{ex} usa um construtor unário para a substituição explicita, mas tem o mesmo poder de expressividade de cálculos com substituições simultâneas.

Palavras-chave: Cálculo λ_{ex}, cálculos de substituições explícitas, Composicionalidade, Confluência

Abstract

We present a formalization of properties of compositionality of the λ_{ex}-calculus in *Coq*. The approach is based in the nominal logic as presented in the paper [3]. More precisely, we extended a formalization of the λ-calculus in such a way that it now includes the explicit substitution operation of the λ_{ex}-calculus. In this approach, α-equivalence of λ-terms coincides with the *Coq*'s built-in equality, and the principles of recursion and induction over classes of λ-terms are treated in a specific way. We chose to work with the λ_{ex}-calculus because it is currently the only calculus that simultaneously satisfies all the desirable properties for a calculus of explicit substitutions. It is an extension of the λ_x-calculus with a rewrite rule for composition of dependent substitutions and one equation for independent substitutions. The λ_{ex}-calculus has a unary constructor for the explicit substitution operation, but have the same expressive power of calculi with simultaneous substitutions.

Keywords: λ_{ex}-calculus, calculi of explicit substitutions, Compositionality, Confluence

Sumário

Capítulo 1

Introdução

Verificação formal é uma técnica que os desenvolvedores podem usar para construir sistemas seguros e livres de erros. É baseada em métodos formais, os quais são matematicamente baseados em linguagens, técnicas e ferramentas para especificar e verificar sistemas. É uma área em crescimento que vem recebendo especial atenção ao longo dos últimos anos [10, 40].

Verificação de *software* via testes (tentativa e erro) não é capaz de provar que um determinado sistema computacional está livre de erros. Somente via verificação formal (isto é, provas matemáticas) é que podemos garantir que um sistema não possui um determinado defeito, ou ainda que possui determinada propriedade. Em contraste com o teste e simulação, métodos formais fornecem a capacidade do estado exaustivo de análise exploratória [11]. Sua utilização é ampla e constitui um importante segmento da área de Computação, incluindo diversas subáreas como Teoria da Computação e Engenharia de Software.

Dentro desse contexto, temos o λ-cálculo que é um sistema formal capaz de expressar qualquer função computável, ou seja, é tão expressivo quanto as máquinas de Turing. Além disso, o λ-cálculo é o formalismo no qual se baseiam as chamadas linguagens funcionais, como por exemplo, Haskell, ML (Ocaml, SML), Lisp, etc. Em outras palavras é o fundamento teórico do paradigma de programação funcional.

Outro ponto relevante são os cálculos de substituições explícitas, amplamente utilizados em diferentes áreas da Ciência da Computação, tais como programação lógica e funcional, teoria da prova, provas de teoremas, linguagens orientadas a objetos, etc. Eles estendem a linguagem do λ-cálculo e internalizam a operação de substituição. Estes formalismos são adequados para o tratamento formal de implementações e sistemas dedutivos.

Provar que um sistema satisfaz determinada propriedade não é, em geral, tarefa simples, por isso utiliza-se um *assistente de provas* chamado *Coq*, que é uma ferramenta que nos auxilia nas provas dos teoremas.

O *Coq*, é uma ferramenta baseada em três linguagens bem coordenadas e complexas. A primeira é uma linguagem de especificação, chamada *Gallina*. A segunda, chamada *Vernacular*, uma liguagem de comandos. Ela permite ao usuário orientar o *Coq* na organização das provas e programas. E a terceira linguagem é composta pelo conjunto de *táticas*, as quais possuem construtores de alto nível que

são utilizados para se transformar, de maneira "semi-automática", provas complexas em provas equivalentes e mais simples.

Neste trabalho estamos interessados no cálculo λ_{ex} [26], que pode ser visto como uma extensão do λ_x [27] , junto com uma regra para substituições dependentes e uma equação de comutatividade de substituições independentes, os quais veremos em mais detalhes nos próximos tópicos, juntamente com suas propriedades.

O foco do trabalho é a formalização dos principais resultados de [27] em *Coq*, dentro do seguinte escopo : a especificação do cálculo λ_{ex} e a prova da confluência. Para se chegar à prova da confluência, formalizamos propriedades importantes do cálculo λ_{ex}, como a composição e composição completa, assim como definimos a propriedade Z, uma função de termos a termos, a qual é definida a partir da função superdesenvolvimento.

A presente dissertação está dividida em sete capítulos, de acordo com a seguinte ordem:

O primeiro capítulo é esta introdução, apresentando o trabalho desenvolvido.

O segundo capítulo contém noções importantes sobre a teoria do λ-cálculo, mostrando seus principais conceitos e definições, tais como a regra de α-conversão, a regra β e a definição de tipos simples.

O terceiro capítulo versa sobre substituicões explícitas. Apresentaremos uma breve visão sobre o assunto, explicando seu significado, e apresentando diversos exemplos e suas propriedades. Além disso, introduzimos neste capítulo, o cálculo λ_{ex} via sua gramática, os principais conceitos e regras de reescrita e concluímos com a prova da confluência. Sobre a confluência, apresentaremos sua prova em detalhes como apresentada em [27].

No quarto capítulo, falaremos do assistente de provas *Coq*, essa importante ferramenta que nos auxiliou neste trabalho, fornecendo uma visão de suas principais características e funcionalidades.

No quinto capítulo introduziremos uma visão geral sobre Lógica Nominal, apresentando seus principais conceitos, tais como : swaps, permutações de átomos, conjuntos, variáveis novas e ligadas. Este assunto tem sido objeto de muitas pesquisas e avanços na área da computação, fornecendo uma nova abordagem para o tratamento de linguagens que possuem ligações (*binders*) para variáveis.

O sexto capítulo aborda a especificação do cálculo λ_{ex} em *Coq*, apresentando diversos conceitos e aplicações de lógica nominal, de acordo com o apresentado no trabalho desenvolvido em [3], realizando uma extensão do mesmo, através da inclusão da substituição explícita. Ainda neste capítulo, e como foco principal de nosso trabalho, apresentaremos os principais resultados da prova da confluência utilizando a definição da propriedade Z [34] e da função superdesenvolvimento.

Por fim, finalizamos o capítulo sete com a conclusão deste trabalho, onde uma tal formalização pode ser útil na construção de sistemas baseados em cálculos de substituições explícitas, e em particular no λ_{ex}, onde uma possível versão em notação de *de Bruijn* [17] é de particular interesse em implementações reais.

Capítulo 2

O λ-Cálculo

Nesta seção faremos uma breve apresentação do λ-cálculo, para maiores detalhes recomendamos a leitura de [4].

O λ-Cálculo é um sistema de reescrita de termos, que foi originalmente desenvolvido por Alonzo Church na década de 30 [12], como parte de uma teoria geral de funções e lógica, objetivando uma fundamentação para a matemática. Todavia o sistema completo continha inconsistências, tal como foi demonstrado por Kleene e Rosser em [28], e assim o subsistema formado somente pela parte da teoria de funções, cuja consistência foi demonstrada por Church [13], se tornou um modelo eficiente para funções computáveis. Este sistema é chamado hoje de λ-cálculo.

Em [29] é demonstrado que todas as funções recursivas podem ser representadas no λ-cálculo. Por outro lado, em [41] é mostrado que exatamente as funções computáveis pela máquina de Turing podem ser representadas no referido cálculo.

Em [6], Barendregt explica o aparecimento do símbolo λ para denotar a abstração de uma função: "Em Principia Mathematica [42], a notação para função f com $f(x) = 2x + 1$ é $2\hat{x} + 1$. Church originalmente pretendia utilizar a notação $\hat{x}.2x + 1$. No entanto o tipógrafo não conseguiu posicionar o símbolo $\hat{}$ em cima da letra x, e o colocou em frente da mesma resultando em $\hat{}x.2x + 1$. Depois outro tipógrafo mudou $\hat{}x.2x + 1$ para $\lambda x.2x + 1$ ".

A gramática do λ-cálculo é baseada em duas operações:

(i) Aplicação: $(a\, b)$

(ii) Abstração: $(\lambda x.a)$
onde $(\lambda x.a)$ representa a função de parâmetro x e corpo a

Os termos do λ-cálculo podem ser definidos por meio da seguinte gramática:

$$M ::= x \mid (M\, M) \mid (\lambda x.M)$$

Isto é, os termos que podem ser construídos no λ-cálculo são compostos somente de variáveis, aplicações e abstrações. Denotaremos por Λ o conjunto de todos os λ-termos.

Por convenção, abstrações sucessivas são associadas à direita, por exemplo, $\lambda x_1.\cdots\lambda x_k.M \equiv (\lambda x_1.\cdots(\lambda x_k.M))$ o que é abreviado por $\lambda x_1 \ldots x_k.M$; e no caso das aplicações, associadas à esquerda: $M_1\ \ M_2\ \ M_3 \ldots M_n \equiv (\ldots((M_1\ \ M_2)\ \ M_3)\ldots M_n)$.

Por exemplo, $((y\ u)\ x)\ ((x\ u)\ (w\ v))$ pode ser escrito como $(yux\ (xu\ (w\ v)))$. Adicionalmente, sempre que não houver ambiguidades de interpretação, os parênteses mais externos também poderão ser omitidos.

Exemplo 2.1. *Aplicação e abstração trabalham juntas. Por exemplo :*
$(\lambda x.2 * x + 1)3 = 2 * 3 + 1 (= 7)$.
Isto é, $(\lambda x.2*x+1)3$ *denota a função* $x \mapsto 2*x+1$ *aplicada ao argumento 3 dando* $2 * 3 + 1$*, o qual é 7. Em geral nós temos* $(\lambda x.M[x])N = M[N]$. *Esta última equação é escrita como :*
$(\lambda x.M)N = M\{x/N\}$,
onde $\{x/N\}$*, denota a substituição de x por N na estrutura de M. Esta substituição será observada mais à frente na regra* β.

A noção de variáveis livres (isto é, variáveis que ocorrem no corpo de uma abstração, mas não tem o mesmo nome de seu parâmetro) é formalizada pela definição abaixo:

Definição 2.2 (Conjunto de variáveis livres de um termo)**.** *Definimos o* ***conjunto das variáveis livres*** *de* a*, denotado por* $FV(a)$*, indutivamente da seguinte forma:*

(i) $FV(x) = x$

(ii) $FV(a\,b) = FV(a) \cup FV(b)$

(iii) $FV(\lambda x.a) = FV(a) \setminus \{x\}$

Nota*: Qualquer variável de "*a*" que não pertence a* $FV(a)$ *é denominada uma* variável ligada de a. *O conjunto das variáveis ligadas do termo* a *é denotado por* $BV(a)$.

Aqui definiremos uma *meta-substituição*, que é uma substituição de ordem superior exterior a gramática do λ-cálculo. Se a convenção de Barendregt for adotada, pode-se realizar a seguinte definição indutiva desse formalismo:

Definição 2.3 (Meta-substituição)**.** *Sendo* a, b *termos e* x, y *variáveis distintas, a* ***meta-substituição*** $a\{x/b\}$ *é definida indutivamente por:*

(i) $x\{x/b\} = b$

(ii) $y\{x/b\} = y$

(iii) $(t\,v)\{x/b\} = (t\{x/b\})\,(v\{x/b\})$

(iv) $(\lambda x.t)\{x/b\} = \lambda x.t$

(v) $(\lambda y.t)\{x/b\} = \lambda y.t\{x/b\}$

Caso não se adote a convenção de Barendregt [4], que diz que: *"Os nomes utilizados para variáveis livres deverão ser sempres distintos dos nomes dados para variáveis ligadas"*, será necessário inserir uma restrição ao item *(v)*, adicionando-se a condição de que $y \notin FV(b)$. Caso contrário pode haver a ocorrência de captura de variáveis livres, gerando contradições no sistema. Adicionalmente, no caso em que $y \in FV(b)$, a nova definição fica da forma abaixo :

$(\lambda_y.t)\{x/b\} = \lambda_z.t\{y/z\}\{x/b\}$ *se* $x \neq y, y \in FV(b)$ *e* $z \notin FV(t\ b)$

Pode-se exemplificar de forma simples, um outro caso de perda semântica por captura de variáveis : o termo $\lambda y.x$ representa uma função constante onde o corpo x não tem nenhuma relação com o parâmetro y. Se aplicarmos a substituição $\{x/y\}$ a este termo de forma "ingênua", isto é como uma substituição de primeira ordem obteremos $\lambda y.y$ que definitivamente possui uma semântica diferente do termo original. Esse fenômeno de captura de variáveis livres, é denominado *colisão entre variáveis livres e ligadas*, e o procedimento correto para aplicar a substituição neste caso utiliza renomeamento de variáveis ligadas: $\lambda y.x\{x/y\} =_\alpha (\lambda z.x)\{x/y\} = (\lambda z.x\{x/y\}) = \lambda z.y$. O processo de renomear variáveis ligadas é conhecido como α-conversão e será descrito a seguir.

A primeira regra do λ-cálculo é a *α-conversão*. Esta estabelece que é permitido fazer renomeamento das variáveis ligadas de um termo, desde que os novos nomes não *colidam* com os nomes de alguma outra variável já utilizada.

Definição 2.4 (Regra de α-conversão)**.** *Sejam x e z variáveis, e $a \in \Lambda$. Então $\lambda x.a =_\alpha \lambda z.a\{x/z\}$, se $z \notin FV(a)$.*
***Nota**: Aqui utiliza-se $=$ ao invés de $\rightarrow$, porque a α-conversão fornece na realidade uma relação de equivalência entre termos.*

Exemplo 2.5. $\lambda x.x\ z =_\alpha \lambda u.(x\ z)\{x/u\} = \lambda u.u\ z$

A noção de computação do λ-cálculo é representada através de sua principal regra, que é chamada β :

Definição 2.6 (Regra β)**.** *Seja x uma variável, M e N termos : $(\lambda x.M)N \rightarrow_\beta M\{x/N\}$*
***Nota**: $M\{x/N\}$ é o termo resultante da substituição simultânea de todas as ocorrências livres de x em M por N, de acordo com a definição indutiva 2.3.*

Exemplo 2.7. $(\lambda xy.xy)y \rightarrow_\beta \lambda z.yz$

Exemplo 2.8. $(\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x) \rightarrow_\beta (\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x)$.

Este exemplo mostra que o λ-Cálculo não é terminante, pois o termo $(\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x)$ reduz para si mesmo. Esse processo pode ser repetido indefinidamente, gerando assim um caminho infinito de redução.

Um primeiro resultado sobre substituições é relatado no seguinte lema:

Lema 2.9 (Lema da Substituição)**.** *Sejam L, M e $N \in \Lambda$ e x, y variáveis. Se $x \neq y$ e $x \notin FV(L)$, então :*

$$M\{x/N\}\{y/L\} = M\{y/L\}\{x/N\{y/L\}\}.$$

Demonstração. Por indução sobre a estrutura de M.
Caso 1 : M é uma variável:
Subcaso 1 : $M = x$. Simplificando os dois termos da igualdade acima, temos :

$$x\{x/N\}\{y/L\} = x\{y/L\}\{x/N\{y/L\}\}$$
$$N\{y/L\} = x\{x/N\{y/L\}\}$$
$$N\{y/L\} = N\{y/L\}.$$

Subcaso 2 : $M = y$ *, com* $y \neq x$

$$y\{x/N\}\{y/L\} = y\{y/L\}\{x/N\{y/L\}\}$$
$$y\{y/L\} = L\{x/N\{y/L\}\}$$
$$L = L.$$

Caso 2 : M *é da forma* $\lambda_z.M'$*. A partir do lado esquerdo da igualdade, temos :*

$$(\lambda_z.M')\{x/N\}\{y/L\} =_{def} \lambda_z.M'\{x/N\}\{y/L\} =_{IH}$$
$$\lambda_z.M'\{y/L\}\{x/N\{y/L\}\} =_{def} (\lambda_z.M')\{y/L\}\{x/N\{y/L\}\}.$$

Caso 3 : M *é da forma* $M1\ M2$*:*

$$(M1\ M2)\{x/N\}\{y/L\} = (M1\{x/N\}\{y/L\})(M2\{x/N\}\{y/L\}) =_{IH}$$
$$(M1\{y/L\}\{x/N\{y/L\}\})(M2\{y/L\}\{x/N\{y/L\}\}) =$$
$$(M1\ M2)\{y/L\}\{x/N\{y/L\}\}.$$

□

Definição 2.10. *As relações binárias* $\rightarrow_\beta$*,* $\xrightarrow{*}_\beta$ *e* $=_\beta$ *em* Λ *são definidas indutivamente como segue :*

- – $(\lambda_x.M)N \rightarrow_\beta M\{x/N\}$ *(um passo de* β*-redução)*
  - – *Se* $M \rightarrow_\beta N$ *então* $Z\ M \rightarrow_\beta Z\ N, M\ Z \rightarrow_\beta N\ Z$ *e* $\lambda_x.M \rightarrow_\beta \lambda_x.N$ *, ou seja, a* β*-redução é compatível com a estrutura dos termos.*
- *O fecho transitivo-reflexivo da* β*-redução é definido por:*
  - – $M \xrightarrow{*}_\beta M$ *(reflexividade)*
  - – *Se* $M \rightarrow_\beta N$ *então* $M \xrightarrow{*}_\beta N$
  - – *Se* $M \xrightarrow{*}_\beta N$ *e* $N \xrightarrow{*}_\beta L$ *então* $M \xrightarrow{*}_\beta L$ *(transitividade)*
- *A relação de equivalência* $=_\beta$ *é definida por :*
  - – *Se* $M \xrightarrow{*}_\beta N$ *então* $M =_\beta N$
  - – *Se* $M =_\beta N$ *então* $N =_\beta M$ *(simetria)*
  - – *Se* $M =_\beta N$ *e* $N =_\beta L$ *então* $M =_\beta L$.

Essas relações são pronunciadas como segue.

$M \xrightarrow{*}_\beta N$: *Se* M *β-reduz a* N;
$M \rightarrow_\beta N$: *Se* M *β-reduz a* N *em um passo*;
$M =_\beta N$: *Se* M *é β-convertível a* N.

Vistas algumas informações relevantes sobre o λ-Cálculo, faremos uma apresentação sobre a confluência.

Definição 2.11. *(Confluência) Um sistema de reescrita* $(M, \rightarrow)$ *é confluente se e somente se, dados quaisquer termos* $u, v, w \in M$ *tais que* $v \ ^* \leftarrow u \rightarrow^* w$*, existe algum* $r \in M$ *com* $v \rightarrow^* r \ ^* \leftarrow w$ *(ou simplesmente* $v \downarrow w$*). Utilizando uma notação compacta, podemos expressar a confluência por:*

$$(^*\leftarrow \circ \rightarrow^*) \subseteq (\rightarrow^* \circ \ ^*\leftarrow)$$

Ou ainda por:

$$(^*\leftarrow \circ \rightarrow^*) \subseteq \downarrow$$

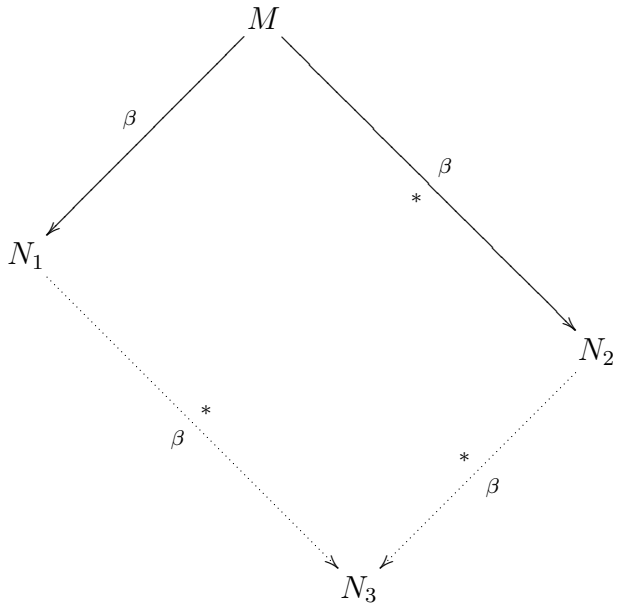
O teorema a seguir é o principal resultado desta seção.

Teorema 2.12.

- *O* λ*-cálculo é confluente;*

A fim de provar o teorema, é suficiente mostrar o Strip Lemma, em que a prova apresentada a seguir foi originalmente obtida em [5].

Lema 2.13 (Strip Lemma)**.** *Sejam* $M, N_1, N_2, N_3 \in \Lambda$*. Se* $M \rightarrow_\beta N_1$ *e* $M \rightarrow_\beta^* N_2$*. Então existe um termo* N_3*, tal que* $N_1 \rightarrow_\beta^* N_3$ *e* $N_2 \rightarrow_\beta^* N_3$ *.*



$M, N_1, N_2, N_3 \in \Lambda$.

Prova do Strip Lemma

Para provar este lema precisaremos apresentar alguns resultados preliminares. Um dos objetivos destes resultados é mostrar como é possível controlar as β-reduções a serem realizadas. Para isto, define-se o conjunto $\underline{\Lambda}$, que tem como propósito marcar os β-rédices que serão normalizados durante as reduções.

Definição 2.14. .

I. $\underline{\Lambda}$ *é definido indutivamente como segue :*

$x \in V \Rightarrow x \in \underline{\Lambda}$; *onde* V *é o conjunto de variáveis.*
$M, N \in \underline{\Lambda} \Rightarrow (MN) \in \underline{\Lambda}$;
$M \in \underline{\Lambda}, x \in V \Rightarrow (\lambda x.M) \in \underline{\Lambda}$;
$M, N \in \underline{\Lambda}, x \in V \Rightarrow ((\underline{\lambda} x.M)N) \in \underline{\Lambda}$

II. Redução (em um passo) $\longrightarrow \underline{\beta}$ *é definida por :*

$(\lambda x.M)N \to M\{x/N\}$,
$(\underline{\lambda} xM)N \to M\{x/N\}$

Então $\longrightarrow \beta$ *é estendida para a relação compatível* $\longrightarrow \underline{\beta}$ *,*
sendo que $\xrightarrow{*} \underline{\beta}$ *é o fecho reflexivo transitivo de* $\longrightarrow \underline{\beta}$ *.*

III. Se $M \in \underline{\Lambda}$*, então* $|M| \in \Lambda$ *é obtido de* M *pela remoção dos sobrescritos.*

IV. A substituição em $\underline{\Lambda}$ *é definida pela regra na definição usual de substituição, adicionando-se :*

$((\underline{\lambda} x.M)N)\{y/L\} = (\underline{\lambda} x.M\{y/L\})(N\{y/L\})$

Definição 2.15. *A função* $\varphi : \underline{\Lambda} \to \Lambda$ *é definida indutivamente como segue:*

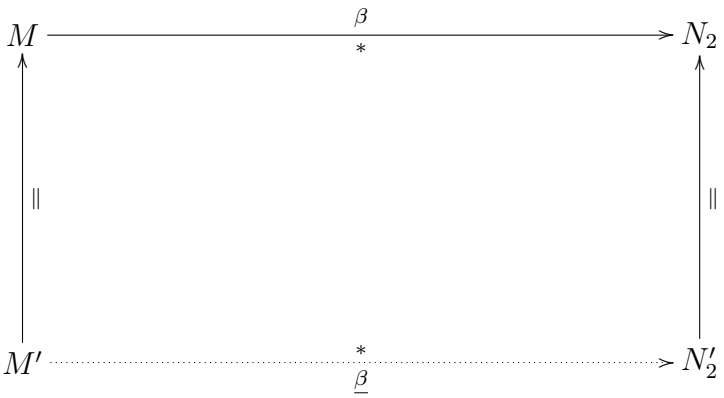
$\varphi(x) = x$;
$\varphi(MN) = \varphi(M)\varphi(N), seM, N \in \underline{\Lambda}$;
$\varphi(\lambda x.M) = \lambda x.\varphi(M)$;
$\varphi((\underline{\lambda} x.M)N) = \varphi(M)\{x/\varphi(N)\}$

Em outras palavras, a função φ *contrai todos os rédices que estão sobrescritos, de dentro para fora.*

Notação 2.16. *Se* $\mid M \mid = N_2$ *ou* $\varphi(M) = N_2$*, então isto será denotado respectivamente por :*

$M \xrightarrow[\|]{} N_2$ *ou* $M \xrightarrow[\varphi]{} N_2$

Lema 2.17. *Sejam* $M', N_2' \in \underline{\Lambda}$ *e* $M, N_2 \in \Lambda$.



Prova. Primeiro suponha $M \xrightarrow[\beta]{} N_2$. *Então* N_2 *é obtido pela contração de um redex em* M, *e* N_2' *pode ser obtido por contrair o redex correspondente em* M'. *Sem perda de generalidade, temos :*

$$M' = (\underline{\lambda}x.P)Q \xrightarrow[\underline{\beta}]{} P\{x/Q\} = N_2'$$
$$M = (\lambda x. \mid P \mid) \mid Q \mid \xrightarrow[\beta]{} \mid P \mid \{x/ \mid Q \mid\} = N_2$$

A afirmação geral segue por transitividade.

□

Lema 2.18. *Sejam* $M, M', N, L \in \underline{\Lambda}$. *Então :*

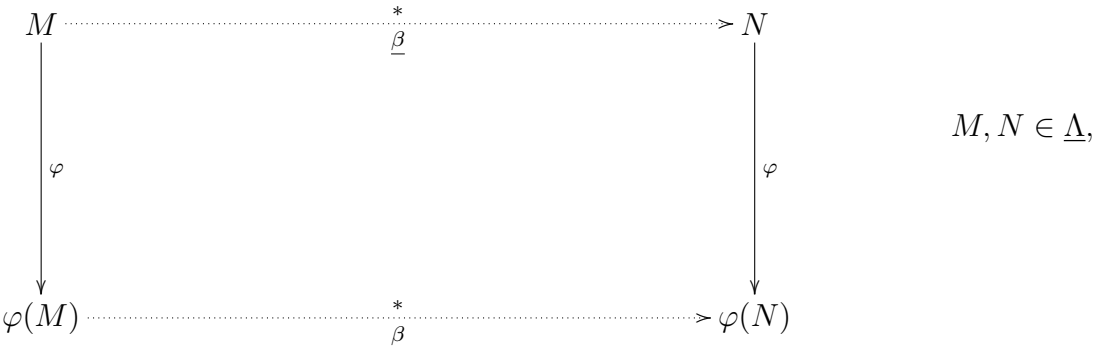
1. Supor $x \neq y$ *e* $x \notin FV(L)$. *Então :*

$$M\{x/N\}\{y/L\} = M\{y/L\}\{x/N\{y/L\}\}$$

Corresponde à generalização do lema da substituição para o $\underline{\Lambda}$

2. $\varphi(M\{x/N\}) = \varphi(M)\{x/\varphi(N)\}$.

3.



Prova.

1. Por indução na estrutura de M *:*
$M = x$:
$x\{x/N\}\{y/L\} = N\{y/L\} = x\{y/L\}\{x/N\{y/L\}\}$;

$M = y$:
$y\{x/N\}\{y/L\} = L = y\{y/L\}\{x/N\{y/L\}\}$;

$M = z, com\ z \neq x\ e\ z \neq y$:
$z\{x/N\}\{y/L\} = z = z\{y/L\}\{x/N\{y/L\}\}$;

$M = (P\ Q)$:
$(P\ Q)\{x/N\}\{y/L\} = (P\{x/N\}Q\{x/N\})\{y/L\} = (P\{x/N\}\{y/L\}Q\{x/N\}\{y/L\})$
$=_{IH}\ (P\{y/L\}\{x/N\{y/L\}\}Q\{y/L\}\{x/N\{y/L\}\}) =$
$(P\ Q)\{y/L\}\{x/N\{y/L\}\}$;

$M = \underline{\lambda}z.P$ *(sendo que* $x \neq z \neq y$*, pela convensão de higiêne de Barendregt):*
$(\underline{\lambda}z.P)\{x/N\}\{y/L\} = \underline{\lambda}z.P\{x/N\}\{y/L\} =_{IH}$
$\underline{\lambda}z.P\{y/L\}\{x/N\{y/L\}\} =$
$(\underline{\lambda}z.P)\{y/L\}\{x/N\{y/L\}\}$;

2. Por indução na estrutura de M*, usando o Lemma da substituição (1) no caso* $M \equiv (\underline{\lambda}y.P)Q$*. A condição de (1) pode ser assumida pela nossa convenção sobre variáveis livres.*

* $M = x$:
$\varphi(x\{x/N\}) = \varphi(N) = x\{x/\varphi(N)\} = \varphi(x)\{x/\varphi(N)\}$;

* $M = y, tal\ que\ y \neq x$:
$\varphi(y\{x/N\}) = \varphi(y) = y\{x/\varphi(N)\} = \varphi(y)\{x/\varphi(N)\}$;

* $M = (P\ Q)$:
$\varphi((P\ Q)\{x/N\}) = \varphi(P\{x/N\}Q\{x/N\}) = \varphi(P\{x/N\})\varphi(Q\{x/N\})$
$=_{IH}\ \varphi(P)\{x/\varphi(N)\}\varphi(Q)\{x/\varphi(N)\}$
$= (\varphi(P)\varphi(Q))\{x/\varphi(N)\} = \varphi(PQ)\{x/\varphi(N)\}$;

* $M = \lambda y.P$:

$\varphi((\lambda y.P)\{x/N\}) = \varphi(\lambda y.P\{x/N\}) = \lambda y.\varphi(P\{x/N\})$
$=_{IH}\ \lambda y.\varphi(P)\{x/\varphi(N)\} = (\lambda y.\varphi(P))\{x/\varphi(N)\}$
$= \varphi(\lambda y.P)\{x/\varphi(N)\}$;

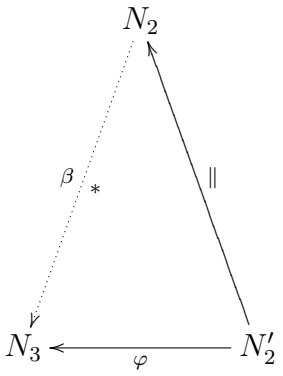
* $M = (\underline{\lambda}y.P)Q$:
$\varphi(((\underline{\lambda}y.P)Q)\{x/N\}) = \varphi((\underline{\lambda}y.P\{x/N\})Q\{x/N\}) = \varphi(P\{x/N\})\{y/\varphi(Q\{x/N\})\}$
$=_{IH} \varphi(P)\{x/\varphi(N)\}\{y/\varphi(Q)\{x/'(N)\}\}$
$(2.) = \varphi(P)\{y/\varphi(Q)\}\{x/\varphi(N)\}$
$= \varphi((\underline{\lambda}y.P)Q)\{x/\varphi(N)\}.$

3. Por indução na geração de $\overset{*}{\underset{\beta}{\dashrightarrow}}$ *, usando (2).*

$\varphi(M\{x/N\}) = \varphi(M)\{x/\varphi(N)\}.$

□

Lema 2.19. *Sejam* $N_2, N_3 \in \Lambda$ *e* $N_2' \in \underline{\Lambda}$.



$N_2' \in \underline{\Lambda}$,

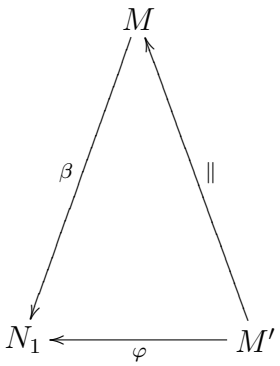
$N_2, N_3 \in \Lambda$.

Prova. Por indução na estrutura de N_2' *:*

$N_2' = x$:
$N_2 =\mid x \mid = x$ *e* $N_3 = \varphi(x) = x$*, portanto* $N_2 \overset{*}{\underset{\beta}{\dashrightarrow}} N_3$ *;*

$N_2' = (PQ)$:
$N_2 =\mid (PQ) \mid = (\mid P \mid\mid Q \mid)$ *e por* IH :
$\mid P \mid \overset{*}{\underset{\beta}{\dashrightarrow}} \varphi(P)$ $\qquad\qquad$ $\mid Q \mid \overset{*}{\underset{\beta}{\dashrightarrow}} \varphi(Q)$
então pela definição de $\overset{*}{\underset{\beta}{\dashrightarrow}}$ *e* φ*, temos que*
$N_2 = \ (\mid P \mid\mid Q \mid) \overset{*}{\underset{\beta}{\dashrightarrow}} (\varphi(P)\varphi(Q)) = N_3$ *;*

$N_2' = \underline{\lambda}x.P$ *:*
$N_2 =\mid \underline{\lambda}x.P \mid = \underline{\lambda}x. \mid P \mid$ *e por IH* $\mid P \mid \overset{*}{\underset{\beta}{\dashrightarrow}} \varphi(P)$ *. então pela definição de* $\overset{*}{\underset{\beta}{\dashrightarrow}}$ *e* φ*, temos que* $N_2 = \ \lambda x \mid P \mid \overset{*}{\underset{\beta}{\dashrightarrow}} \lambda x\varphi(P) = N_3$ *.*



$M' \in \underline{\Lambda}$,

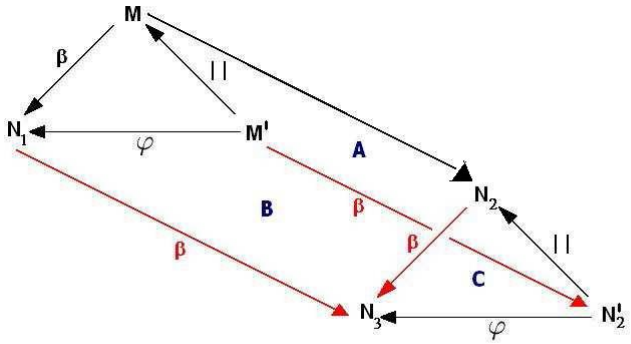
$M, N_1 \in \Lambda$.

Prova. Por indução na estrutura de M', aplicando-se o raciocínio da prova anterior.

□

E concluindo a prova do strip lemma, temos que :

Em N_1 está o resultado da contração da ocorrência do redex $R = (\lambda x.P)Q$ em M. O termo $M' \in \underline{\Lambda}$ é obtido de M pela substituição de R por $R' = (\underline{\lambda} x.P)Q$. Então $|M'| = M$ e $\varphi(M') = N_1$. Pelos Lemas: 2.17, 2.18 e 2.19 nós podemos construir o seguinte diagrama, o que conclui a prova do strip lemma.



□

Definição 2.20. *Os **tipos simples** são indutivamente definidos como segue:*

(i) $\alpha, \alpha', ... \in T$ *(variáveis de tipo);*

(ii) $\sigma, \tau \in T \Rightarrow (\sigma \to \tau) \in T$ *(tipos funcionais).*

As regras de inferência para termos simplesmente tipados são dadas por:

(axioma) $\dfrac{}{\Gamma \vdash x : \sigma}$ se $(x : \sigma) \in \Gamma$

(→ - eliminação) $\dfrac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash (t\, u) : \tau}$

(→ - introdução) $\dfrac{\Gamma, (x : \sigma) \vdash t : \tau}{\Gamma \vdash (\lambda x.\, t) : \sigma \to \tau}$

O cálculo $\lambda_{\rightarrow}$ é formado pelos termos que podem ser construídos utilizando-se as regras de dedução acima. Um outro exemplo interessante desse sistema de tipos é a derivação, conforme observa-se abaixo :

$$\dfrac{\dfrac{\dfrac{(x : Bool) \in \{x : Bool\}}{x : Bool \vdash x : Bool}(\text{axioma})}{\vdash \lambda x : Bool.x : Bool \rightarrow Bool}(\rightarrow \text{ - introdução}) \qquad \vdash true : Bool}{\vdash (\lambda x : Bool.x)\ true : Bool}(\rightarrow \text{ - eliminação})$$

Existem atualmente duas importantes abordagens para introduzir tipos no λ-cálculo : (i) chamada estilo Church ou algumas vezes explícito ou rígido, e (ii) chamado estilo Curry ou algumas vezes implícito [20, 21].

É interessante observar que todo termo simplesmente tipado no λ-cálculo é fortemente normalizável, e além disso, há a ocorrência do chamado isomorfismo de *Curry-Howard*, que aponta a correspondência entre construção de provas em lógica implicacional e a construção de λ-termos simplesmente tipados. Por exemplo, o *tipo funcional*: `nat -> nat` pode ser dado ao termo $(\lambda x.x)$, que corresponde a função identidade, que recebe números naturais como argumento e retorna um número natural. O mesmo acontece quando fornecemos o tipo `Z -> Z` a um certo termo, para que este tenha como resultado o comportamento de uma função de inteiros para inteiros.

Originalmente o paradigma da tipagem implícita foi introduzido por [16] para a teoria de combinadores. Em [15] a teoria foi modificada de uma maneira natural no cálculo λ, designando-se elementos de um dado conjunto T de termos de tipos simples. Por esta razão, este cálculo à la Curry algumas vezes chamado de designação de tipos simples [4]. Se o tipo $\sigma \in T$ é associado ao λ-termo a, denota-se $\vdash a : \sigma$. Usualmente um conjunto de designações de variáveis Γ é necessário para derivar uma designação de tipo, e neste caso pode-se escrever: $\Gamma \vdash a : \sigma$. Um sistema particular, de designação de tipos à la Curry, depende de dois parâmetros: o conjunto T e as regras de inferência de tipos. O λ-cálculo simplesmente tipado é denotado por $\lambda_{\rightarrow}$.

Capítulo 3

Cálculos de Substituições Explícitas

Cálculos de substituições explícitas são utilizados em diferentes áreas da ciência da computação, tal como programação lógica e funcional, teoria da prova, prova de teoremas, linguagens orientadas a objeto etc. Sistemas complexos com substituições explícitas foram desenvolvidos nestes últimos 25 anos, com objetivo de fornecer um formalismo adequado e mais próximo das implementações para se analisar propriedades de sistemas baseados no λ-cálculo [25].

Os cálculos de substituições explícitas são extensões do λ-cálculo, cujo principal objetivo é a obtenção de um cálculo, onde a operação de substituição é removida do nível metalinguístico e inserida na gramática do formalismo em questão.

No λ-cálculo, o processo de avaliação é modelado pela β-redução e a troca de parâmetros formais, por seus correspondentes argumentos, é modelada pela substituição. Enquanto que a substituição no λ-cálculo é uma meta-operação, em cálculos de substituições explícitas este processo é internalizado e manipulado por símbolos e regras de redução, dentro da própria sintaxe do sistema.

Dado que existem muitas dessas extensões, que implementam substituições explícitas, há de se verificar que os novos formalismos gerados atendem a determinados objetivos. Uma teoria de substituições explícitas para o λ-cálculo que possui todas as propriedades esperadas, tal como a *simulação de um passo da β-redução*, a *confluência em metatermos*, a *preservação da β-normalização forte*, a *normalização forte de termos simplesmente tipados* e *composição completa*, é algo raro.

Mais precisamente, seja λ_z um cálculo com substituições explícitas, e considerando uma função (f_λ) da sintaxe λ para a sintaxe λ_z (algumas vezes esta função é justamente a identidade). Deverão ser pesquisadas as seguintes propriedades:

- (C) A relação de redução definida por λ_z é confluente;
- (MC) A relação de definida por λ_z é confluente nos termos abertos. A confluência em metatermos é desejável, dado que muitas vezes precisamos completar provas que contenham *lacunas*. Tal estratégia é muitas vezes utilizada também em problemas de unificação, e generalizações nas demonstrações;

- (PSN) A relação de redução λ_z preserva a β-normalização-forte, isto é, termos fortemente normalizantes no λ-cálculo, também não possuem cadeias de redução infinitas no cálculo λ_z;
- (SN) A normalização forte vale para os termos tipados do λ_z : Se t é tipado, então $t \in SN_{\lambda_z}$.
- (SIM) Um passo da β-redução no λ-cálculo pode ser simulado por λ_z. Ou seja, se $t \to_\beta t'$ então $f_\lambda(t) \to^*_{\lambda_z} f_\lambda(t')$.
- (FC) Full composition pode ser implementada por λ_z : $t[x/u]$ λ_z-reduz $t\{x/u\}$ para uma apropriada noção de substituição nos termos do λ_z.

Adaptamos aqui uma tabela em [25] que aponta a presença ou não, das propriedades acima mencionadas, para alguns cálculos de substituições explícitas:

Cálculo	*C*	*MC*	*PSN*	*SN*	*SIM*	*FC*
λ_x[31, 32, 39]	*Sim*	*Não*	*Sim*	*Sim*	*Sim*	*Não*
λ_{ex}[27]	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*
λ_σ[1]	*Sim*	*Sim*	*Não*	*Sim*	*Sim*	*Sim*
λ_{se}[23, 24]	*Sim*	*Sim*	*Não*	*Sim*	*Sim*	*Sim*
λ_s[23, 24]	*Sim*	*Não*	*Sim*	*Sim*	*Sim*	*Não*

Figura 3.1. Quadro comparativo

3.1 O Cálculo λ_{ex}

Uma maneira natural de especificar o λ-cálculo, com substituições explícitas, é a codificação explícita da definição 2.3, tal que ela continue funcionando via α-conversão, o que a grosso modo, resulta no cálculo λ_x [9, 31, 32, 39]:

Definição 3.1 (Cálculo λ_x)**.** *:*

- *Gramática:*

$$\mathcal{T}_x ::= x \mid (\mathcal{T}_x\, \mathcal{T}_x) \mid (\lambda x.\mathcal{T}_x) \mid \mathcal{T}_x[x/\mathcal{T}_x]$$

- *Regras de Reescrita:*

$$\begin{array}{llll} (\lambda x.t)u & \to_B & t[x/u] & \\ x[x/u] & \to_{Var} & u & \\ y[x/u] & \to_{Gc} & y & \textit{se } x \neq y \\ (tu)[x/v] & \to_{App} & t[x/v]\, u[x/v] & \\ (\lambda y.t)[x/v] & \to_{Lamb} & \lambda y.t[x/v] & \end{array}$$

Este sistema de redução pode ser visto como o mínimo desenvolvimento necessário para se incluir a operação de substituição de forma explícita. Entretanto, a implementação do operador de substituição explícita tem um preço. De fato,

enquanto o λ-cálculo é confluente, o cálculo definido pelas regras acima sofre de no mínimo um bem conhecido exemplo de divergência não juntável [25]:

$$t[y/v][x/u[y/v]] \stackrel{*}{\leftarrow} ((\lambda x.t)u)[y/v] \stackrel{*}{\rightarrow} t[x/u][y/v] \tag{3.1}$$

Diferentes soluções são adotadas com intuito de fechar este diagrama. Uma solução é a adição da regra de reescrita:

$$t[x/u][y/v] \quad \rightarrow_{Comp} \quad t[y/v][x/u[y/v]] \quad \textit{se } y \in \mathrm{FV}(u) \; e \; x \notin \mathrm{FV}(v) \tag{3.2}$$

Dessa maneira, (3.2) adicionada à Definição 3.1 permite fechar o diagrama acima, e mais ainda, garante a confluência do cálculo. A restrição $y \in \mathrm{FV}(u)$ é feita, para separar as substituições dependentes e independentes em dois casos distintos. De fato, se observarmos somente o sistema x (sistema formado por todas as regras da Definição 3.1, exceto a regra B) e não inserirmos a restrição $y \in \mathrm{FV}(u)$ à aplicação da regra (3.2), poderemos ter um caso de não terminação, quando $y \notin \mathrm{FV}(u)$ e $x \notin \mathrm{FV}(v)$:

$$t[x/u][y/v] \rightarrow_{Comp} t[y/v][x/\underline{u[y/v]}] \rightarrow_{Gc} t[y/v][x/u] \rightarrow_{Comp} t[x/u][y/\underline{v[x/u]}] \rightarrow_{Gc} t[x/u][y/v] \rightarrow_{Comp} \ldots$$

A regra (3.2) nada mais é que uma regra de composição para substituições explícitas. Esse tipo de regra aparece primeiramente no λ_σ [1], e dentre outras coisas é importante para estabelecer a confluência em termos abertos [19, 25]. Infelizmente o λ_x não é confluente em termos abertos (termos com meta-variáveis), o que é uma deficiência desse sistema.

Uma outra propriedade importante, além da confluência em termos abertos, é a chamada *composição completa*, que diz que: qualquer termo da forma $t[x/u]$ pode ser reduzido para $t\{x/u\}$; em outras palavras, substituições explícitas implementam substituições implicitas.

Muitos cálculos de substituições explícitas tal como o λ_σ, $\lambda_{\sigma_\Uparrow}$ [19], têm composição completa. Todavia, o λ_σ e o $\lambda_{\sigma_\Uparrow}$ não preservam a normalização forte como constatado em [33]. Entretanto a composição completa e normalização forte podem ocorrer juntas; este é, por exemplo, o caso no cálculo λ_{es} [25].

A equação $\boldsymbol{C}$, adicionada a composição, fornece a definição do sistema de regras do cálculo λ_{ex} [27]:

Equação:			
$t[x/u][y/v]$	$=_C$	$t[y/v][x/u]$	*se* $y \notin \mathbf{FV}(u)$ & $x \notin \mathbf{FV}(v)$
Regras:			
$(\lambda x.t)u$	$\rightarrow_B$	$t[x/u]$	
$x[x/u]$	$\rightarrow_{Var}$	u	
$t[x/u]$	$\rightarrow_{Gc}$	t	*se* $x \notin \mathbf{FV}(t)$
$(tu)[x/v]$	$\rightarrow_{App}$	$t[x/v]\, u[x/v]$	
$(\lambda y.t)[x/v]$	$\rightarrow_{Lamb}$	$\lambda y.t[x/v]$	
$t[x/u][y/v]$	$\rightarrow_{Comp}$	$t[y/v][x/u[y/v]]$	*se* $y \in \mathbf{FV}(u),\ x \notin \mathbf{FV}(v)$

Figura 3.2. Cálculo λ_{ex}

Variáveis livres e ligadas são definidas do modo usual: a variável x ocorre ligada em $\lambda x.t$ e em $t[x/u]$. Então assumindo α-conversão, temos por exemplo: $(\lambda y.x)[x/y] =_\alpha (\lambda z.x)[x/y]$.

A definição de meta-substituições em x-termos é muito semelhante a dada pela *Definição 2.3*. Na realidade, todos os itens daquela definição são incluídos nesta, e ainda é acrecescentado um item para o caso onde o x-termo é uma *closure*: $t[y/u]\{x/v\}$.

Definição 3.2 (meta-substituição para x-termos)**.** *Sendo u, v termos e x, y variáveis distintas, a **meta-substituição** $u\{x/v\}$ é definida indutivamente por:*

(i) $x\{x/v\} = v$
(ii) $y\{x/v\} = y$
(iii) $(t\, v)\{x/v\} = (t\{x/v\})\,(v\{x/v\})$
(iv) $(\lambda x.t)\{x/v\} = \lambda x.t$
(v) $(\lambda y.t)\{x/v\} = \lambda y.t\{x/v\},\ se\ y \notin FV(v)$
(vi) $(\lambda y.t)\{x/v\} = \lambda z.t\{y/z\}\{x/v\},\ se\ y \in FV(v),\ onde\ z \notin FV(t\, v)$
(vii) $t[y/u]\{x/v\} = t\{x/v\}[y/u\{x/v\}],\ se\ y \notin FV(v)$
(viii) $t[y/u]\{x/v\} = t\{y/z\}\{x/v\}[z/u\{x/v\}],\ se\ y \in FV(v),\ onde\ z \notin FV(t\, v\, u)$

Nota: Toda aplicação da definição da meta-substituição, é realizada pelo módulo α-conversão.

A inclusão de *tipos simples* para x-termos é realizada da maneira usual considerando-se as seguintes regras de inferência de tipos:

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

$$\frac{\Gamma,\, x : A \vdash t : B}{\Gamma \vdash (\lambda x.\, t)\ : A \rightarrow B} \qquad \frac{\Gamma \vdash u : B \quad \Gamma, x : B \vdash t : A}{\Gamma \vdash t[x/u]\ : A}$$

Como já dissemos anteriormente, o cálculo-λ_{ex} satisfaz todas as propriedades desejáveis para um cálculo de substituições explícitas, e como objeto de nosso estudo, escolhemos formalizar a confluência, por considerá-la uma das mais relevantes propriedades. No tópico seguinte apresentamos a formalização da confluência nos metatermos, partindo da prova em papel e lápis, e posteriormente, no capítulo 6, utilizando o assistente de prova *Coq*.

3.2 Composicionalidade

Nesta subseção, faremos um estudo da confluência do cálculo-λ_{ex} sobre os metatermos, que são termos contendo metavariáveis, e no capítulo sobre a especificação do Cálculo λ_{ex} em *Coq*, faremos a formalização mecânica, considerando apenas os termos, ficando os metatermos para atividades futuras.

O estudo inicia-se com a definição do conjunto dos metatermos, especificado pela seguinte gramática :

$$M ::= x \mid X_\Delta \mid MM \mid \lambda_x.M \mid M[x/M]$$

Observe que termos são em particular metatermos, onde X_Δ é uma metavariável, e Δ é o conjunto de variáveis que pode ocorrer em X_Δ.

Meta-substituição nos metatermos estende-se nos termos por adição de dois novos casos :

- $X_\Delta\{x/v\} = X_\Delta \;\; se \;\; x \notin \Delta$
- $X_\Delta\{x/v\} = X_\Delta[x/v] \;\; se \;\; x \in \Delta$

A propriedade abaixo nos mostra que, se x não pertence às variáveis livres de t, então a substituição não terá efeito :

Lema 3.3. *[27] Sejam t, u metatermos. Então $t\{x/u\} = t$ se $x \notin FV(t)$.*

Prova. Por indução em t.

- $t = z$ (t é uma variável)
  $z \neq x$. $\quad z\{x/u\} =_{def} z$.

- $t = Z_\Delta$ (t é uma meta-variável)
  Por hipótese temos que $x \notin Z_\Delta$, e portanto $x \notin \Delta$. Daí :
  $Z_\Delta\{x/u\} =_{def} Z_\Delta$.

- $t = (t1 \;\; t2)$. Seja x uma variável, temos :
  $(t1 \;\; t2)\{x/u\} =_{def} t1\{x/u\} \;\; t2\{x/u\}$.
  Por Hipótese de indução, temos que :
  $t1\{x/u\} = t1 \;\; pois \;\; x \notin FV(t1)$ *e*
  $t2\{x/u\} = t2 \;\; pois \;\; x \notin FV(t2)$.
  Uma vez que $x \notin FV(t1 \;\; t2)$, temos que $(t1 \;\; t2)\{x/u\} = t1\{x/u\} \; t2\{x/u\} = t1 \; t2$.

- $t = (\lambda_y.t')$, temos que :
  $(\lambda_y.t')\{x/u\} =_{def} \lambda_y.t'\{x/u\}$, se $y \notin FV(u)$. Daí, por IH, temos que $t'\{x/u\} = t'$, e portanto : $\lambda_y.t'\{x/u\} = \lambda_y.t'$. Caso $y \in FV(u)$, temos :

$(\lambda_y.t')\{x/u\} =_{def+\alpha} \lambda_z.t'\{y/z\}\{x/u\} =_{H.I} \lambda_z.t'\{y/z\} =_\alpha \lambda_y.t'$.

- $t = s[y/v]$, temos que :
  $s[y/v]\{x/u\} =_{def} s\{x/u\}[y/v\{x/u\}] =_{IH} s[y/v]$, pois $x \notin FV(s\ v)$.

  Temos $x \notin FV(s[y/v]) = FV(s) \cup FV(v)$.

  Caso $y \notin FV(u)$:

  $s[y/v]\{x/u\} = s\{x/u\}[y/v\{x/u\}] =_{IH} s[y/v]$.

  Caso $y \in FV(u)$:

  $s[y/v]\{x/u\} = s\{y/z\}\{x/u\}[z/v\{x/u\}]$, como $x \notin FV)s)$ e $z \notin FV(s)$ então $x \notin FV(s\{y/z\})$. Daí : $s\{y/z\}\{x/u\} = s\{y/z\}$ e também $v\{x/u\} = v$.

  Logo $s\{y/z\}\{x/u\}[z/v\{x/u\}] =_{IH} \ s\{y/z\}[z/v] =_\alpha s[y/v]$.

□

A relação de equivalência gerada pelas α conversões e C é denotada por $=_e$. Pode-se mostrar por indução nos metatermos a seguinte propriedade abaixo :

Lema 3.4. *[27](Composição) Sejam t, u, v metatermos e x, y tal que $x \neq y$ e $x \notin FV(v)$. Então $t\{x/u\}\{y/v\} \ =_e \ t\{y/v\}\{x/u\{y/v\}\}$.*

Prova. Por indução nos metatermos.

- $t = z$ (t é uma variável). Consideraremos 3 casos :
  1. $z \neq x \ \ e \ \ z \neq y$.
  Do lado esquerdo da equação, temos :
  $z\{x/u\}\{y/v\} =_{def} \ z\{y/v\} =_{def} \ z$.
  Do lado direito da equação, temos :
  $z\{y/v\}\{x/u\{y/v\}\} =_{def} \ z\{x/u\{y/v\}\} =_{def} \ z$.

  2. $t = x \ \ e \ \ z \neq y$.
  Do lado esquerdo da equação, temos :
  $x\{x/u\}\{y/v\} =_{def} \ u\{y/v\}$.
  Do lado direito da equação, temos :
  $x\{y/v\}\{x/u\{y/v\}\} =_{def} \ x\{x/u\{y/v\}\} =_{def} \ u\{y/v\}$.

  3. $z \neq x \ \ e \ \ z = y$.
  Do lado esquerdo da equação, temos :
  $y\{x/u\}\{y/v\} =_{def} \ y\{y/v\} =_{def} \ v$.
  Do lado direito da equação, temos :
  $y\{y/v\}\{x/u\{y/v\}\} =_{def} \ v\{x/u\{y/v\}\} =_{def} \ v$.

- $t \equiv Z_\Delta$ (t é uma meta-variável). Consideraremos 4 casos :
  1. $x \notin \Delta \;\; e \;\; y \notin \Delta$.
  Do lado esquerdo da equação, temos :
  $Z_\Delta\{x/u\}\{y/v\} =_{def} \;\; Z_\Delta\{y/v\} =_{def} \;\; Z_\Delta$.
  Do lado direito da equação, temos :
  $Z_\Delta\{y/v\}\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta$.

  2. $x \notin \Delta \;\; e \;\; y \in \Delta$.
  Do lado esquerdo da equação, temos :
  $Z_\Delta\{x/u\}\{y/v\} =_{def} \;\; Z_\Delta\{y/v\} =_{def} \;\; Z_\Delta[y/v]$.
  Do lado direito da equação, temos :
  $Z_\Delta\{y/v\}\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta[y/v]\{x/u\{y/v\}\} = Z_\Delta\{x/u\{y/v\}\}[y/v]$, pois $x \notin FV(v)$. Daí, por definição, temos : $Z_\Delta[y/v]$

  3. $x \in \Delta \;\; e \;\; y \notin \Delta$.
  Do lado esquerdo da equação, temos :
  $Z_\Delta\{x/u\}\{y/v\} =_{def} \;\; Z_\Delta[x/u]\{y/v\} =_{def} \;\; Z_\Delta\{y/v\}[x/u\{y/v\}] =_{def} Z_\Delta[x/u\{y/v\}]$.
  Do lado direito da equação, temos :
  $Z_\Delta\{y/v\}\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta[x/u\{y/v\}]$.

  4. $x \in \Delta \;\; e \;\; y \in \Delta$.
  Do lado esquerdo da equação, temos :
  $Z_\Delta\{x/u\}\{y/v\} =_{def} \;\; Z_\Delta[x/u]\{y/v\} =_{def} \;\; Z_\Delta\{y/v\}[x/u\{y/v\}] =_{def}$
  $Z_\Delta[y/v][x/u\{y/v\}]$.
  Do lado direito da equação, temos :
  $Z_\Delta\{y/v\}\{x/u\{y/v\}\} =_{def} \;\; Z_\Delta[y/v]\{x/u\{y/v\}\} = Z_\Delta[y/v][x/u\{y/v\}]$, pois $x \notin FV(v)$. Então : $Z_\Delta\{x/u\{y/v\}\}[y/v] =_{def} Z_\Delta[x/u\{y/v\}][y/v] =_e Z_\Delta[y/v][x/u\{y/v\}]$.

- $t = (t1 \;\; t2)$.
  $(t1 \;\; t2)\{x/u\}\{y/v\} =_{def} t1\{x/u\}\{y/v\} \;\; t2\{x/u\}\{y/v\}$
  Por Hipótese de indução, temos que :
  $t1\{x/u\}\{y/v\} = t1\{y/v\}\{x/u\{y/v\}\}$.
  $t2\{x/u\}\{y/v\} = t2\{y/v\}\{x/u\{y/v\}\}$.
  Então, $t1\{x/u\}\{y/v\} \;\; t2\{x/u\}\{y/v\} =_{IH} \;\; t1\{y/v\}\{x/u\{y/v\}\} \;\; t2\{y/v\}\{x/u\{y/v\}\} = (t1 \; t2)y/vx/uy/v$.

- $t = (\lambda_z.t') \;\; se \;\; z \notin (FV(u) \;\; \cup \;\; FV(v))$.
  $(\lambda_z.t')\{x/u\}\{y/v\} =_{def} \lambda_z.t'\{x/u\}\{y/v\} =_{IH}$

$\lambda_z.t'\{y/v\}\{x/u\{y/v\}\} =_{def} \quad (\lambda_z.t')\{y/v\}\{x/u\{y/v\}\}$.

- $t = (\lambda_z.t')$ *se* $z \in (FV(u) \ \cup \ FV(v))$.
  $(\lambda_z.t')\{x/u\}\{y/v\} =_{def} \lambda_w.t'\{z/w\}\{x/u\}\{y/v\} =_{IH}$
  $\lambda_w.t'\{z/w\}\{y/v\}\{x/u\{y/v\}\} =_{def} (\lambda_w.t'\{z/w\})\{y/v\}\{x/u\{y/v\}\} =_{\alpha}$
  $(\lambda_z.t')\{y/v\}\{x/u\{y/v\}\}$.
- $t = s[z/w]$.
  $s[z/w]\{x/u\}\{y/v\}$, *temos* $x \notin FV(v)$.
  $Caso\ 1\ :\ z \notin FV(u)\ e\ z \notin FV(v)$.
  $s[z/w]\{x/u\}\{y/v\} =_{def} s\{x/u\}[z/w\{x/u\}]\{y/v\} =_{def} s\{x/u\}\{y/v\}[z/w\{x/u\}\{y/v\} =_{IH}$
  $s\{y/v\}\{x/u\{y/v\}\}[z/w\{y/v\}\{x/u\{y/v\}] = s[z/w]\{y/v\}\{x/u\{y/v\}\}$ *pois* $z \notin FV(u\ v)$.

  $Caso\ 2\ :\ z \in FV(u)\ e\ z \notin FV(v)$.
  $s[z/w]\{x/u\}\{y/v\} =_{def} s\{z/z'\}\{x/u\}[z'/w\{x/u\}]\{y/v\}$, **onde** $z' \notin FV(s\ w\ u\ v)$.
  Daí, temos :
  $s\{z/z'\}\{x/u\}[z'/w\{x/u\}]\{y/v\} =_{def} s\{z/z'\}\{x/u\}\{y/v\}[z'/w\{x/u\}\{y/v\}] =_{IH}$
  $s\{z/z'\}\{y/v\}\{x/u\{y/v\}[z'/w\{y/v\}\{x/u\{y/v\}\}] =_{def}$
  $s\{z/z'\}[z'/w]\{y/v\}\{x/u\{y/v\}\} =_{\alpha} s[z/w]\{y/v\}\{x/u\{y/v\}\}$. **Pois** $z' \notin FV(u\ v)$.

  $Caso\ 3\ :\ z \notin FV(u)\ e\ z \in FV(v)$.
  $s[z/w]\{x/u\}\{y/v\} =_{def} s\{x/u\}[z/w\{x/u\}]\{y/v\}$.
  como $z \in FV(v)$ **então, seja** $z' \notin FV(s\ u\ v\ w)$.
  $s\{x/u\}[z/w\{x/u\}]\{y/v\} =_{\alpha} s\{x/u\}\{z/z'\}[z'/w\{x/u\}]\{y/v\} =_{def}$
  $s\{x/u\}\{z/z'\}\{y/v\}[z'/w\{x/u\}\{y/v\}] = s\{z/z'\}\{x/u\}\{y/v\}[z'/w\{x/u\}\{y/v\}]$, **pois** $z \notin FV(u)$.
  Por hipótese de indução :
  $s\{z/z'\}\{x/u\}\{y/v\}[z'/w\{x/u\}\{y/v\}] =_{IH}$
  $s\{z/z'\}\{y/v\}\{x/u\{y/v\}\}[z'/w\{y/v\}\{x/u\{y/v\}\}] =_{def}$, **pois** $z' \notin FV(u\ v)$,
  $s\{z/z'\}[z/w]\{y/v\}\{x/u\{y/v\}\} =_{\alpha} s[z/w]\{y/w\}\{x/u\{y/v\}\}$.

  $Caso\ 4\ :\ z \in FV(u)\ e\ z \in FV(v)$.
  $s[z/w]\{x/u\}\{y/v\} =_{\alpha} s\{z/z'\}[z'/w]\{x/u\}\{y/v\}$, **onde** $z' \notin FV(s\ w\ u\ v)$. **Por definição :**
  $s\{z/z'\}[z'/w]\{x/u\}\{y/v\} =_{def} s\{z/z'\}\{x/u\}[z'/w\{x/u\}]\{y/v\} =_{def}$
  $s\{z/z'\}\{x/u\}\{y/v\}[z'/w\{x/u\}\{y/v\}] =_{IH}$
  $s\{z/z'\}\{y/v\}\{x/u\{y/v\}\}[z'/w\{y/v\}\{x/u\{y/v\}\}] =_{def}$
  $s\{z/z'\}[z'/w]\{y/v\}\{x/u\{y/v\}\}$, **pois** $z' \notin FV(uv)$.
  $=_{\alpha} s[z/w]\{y/v\}\{x/u\{y/v\}\}$ □

A redução nos metatermos deve ser compreendida da mesma forma que redução nos termos, onde a relação λ_{ex} é gerada pela relação de redução $\rightarrow_{Bx}$ na classe de equivalência ($=_e$) de metatermos. Redução nos termos e metatermos são compatíveis com a substituição e composição completa, e que será objeto de estudo nas provas a seguir. A propriedade SN declarada no lema seguinte, significa *strong normalization*, onde o termo é fortemente normalizável, se cada sequência de reescrita, a partir de t, é finita.

Lema 3.5. *[27](Propriedades básicas). Sejam t, u metatermos. Para $R \in \{ex, \lambda_{ex}\}$, Se $t \rightarrow_R t'$, então $u\{x/t\} \rightarrow_R^* u\{x/t'\}$ e $t\{x/u\} \rightarrow_R t'\{x/u\}$. Assim, em particular $t\{x/u\} \in SN_R$ implica $t \in SN_R$.*

Prova. Por indução em $t \rightarrow t'$.

No caso em que $u\{x/t\} \rightarrow_R^* u\{x/t'\}$, assumimos, sem perda de generalidade, que a redução ocorre na raíz do termo :

Se $x \notin FV(u)$ então $u\{x/t\} \rightarrow_{\lambda_{ex}}^0 u\{x/t'\} = u \rightarrow_{\lambda_{ex}}^0 u$

Se $x \in FV(u)$, então, considerando o sistema λ_{ex}, temos :
Regra $(\lambda_y.t)\ v \rightarrow_B t[y/v]$, que pela IH $t \rightarrow t'$
$u\{x/(\lambda_y.t)\ v\} \rightarrow_B^+ u\{x/t[y/v]\}$

Regra $y[y/v] \rightarrow_{Var} v$, que pela IH $t \rightarrow t'$
$u\{x/y[y/v]\} \rightarrow_{Var}^+ u\{x/v\}$

Regra $t[z/v] \rightarrow_{Gc} t$, onde $(z \notin FV(t))$, que pela IH $t \rightarrow t'$
$u\{x/t[z/v]\} \rightarrow_{Gc}^+ u\{x/t\}$

Regra $(u1\ u2)[y/v] \rightarrow_{App} u1[y/v]\ u2[y/v]$,que pela IH $t \rightarrow t'$
$u\{x/(u1\ u2)[y/v]\} \rightarrow_{App}^+ u\{x/u1[y/v]\ u2[y/v]\}$

Regra $(\lambda_y.t1)[z/t2] \rightarrow_{Lamb} \lambda_y.t1[z/t2]$, que pela IH $t \rightarrow t'$
$u\{x/(\lambda_y.t1)[z/t2]\} \rightarrow_{Lamb}^+ u\{x/\lambda_y.t1[z/t2]\}$

Regra $t[z/s][y/v] \rightarrow_{Comp} t[y/v][z/s[y/v]]$,onde $y \in FV(s)$, que pela IH $t \rightarrow t'$
$u\{x/t[z/s][y/v]\} \rightarrow_{Comp}^+ u\{x/t[y/v][z/s[y/v]]\}$

Regra $t[z/s][y/v] \rightarrow t[y/v][z/s]$,onde $y \notin FV(s)$ e $z \notin FV(v)$, que pela IH $t \rightarrow t'$
$u\{x/t[z/s][y/v]\} \rightarrow_C^+ u\{x/t[y/v][z/s]\}$

No caso em que $t\{x/u\} \rightarrow_R t'\{x/u\}$, assumimos, sem perda de generalidade, que a redução ocorre na raíz do termo :

Se $x \notin FV(t)$ e $x \notin FV(t')$ então $t\{x/u\} \rightarrow_{\lambda_{ex}} t'\{x/u\} = t \rightarrow_{\lambda_{ex}} t'$

Se $x \in FV(t)$ e $x \in FV(t')$, então, considerando o sistema λ_{ex}, temos :

Regra $(\lambda_y.t)\ v \rightarrow_B t[y/v]$, que pela IH $t \rightarrow t'$
$(\lambda_y.t)\ v\{x/u\} \rightarrow_B t[y/v]\{x/u\}$

Regra $y[y/v] \rightarrow_{Var} v$, que pela IH $t \rightarrow t'$
$y[y/v]\{x/u\} \rightarrow_{Var} v\{x/u\}$

Regra $t[z/v] \rightarrow_{Gc} t$, onde $(z \notin FV(t))$, que pela IH $t \rightarrow t'$
$t[z/v]\{x/u\} \rightarrow_{Gc} t\{x/u\}$

Regra $(u1\ u2)[y/v] \rightarrow_{App} u1[y/v]\ u2[y/v]$,que pela IH $t \rightarrow t'$
$(u1\ u2)[y/v]\{x/u\} \rightarrow_{App} u1[y/v]\ u2[y/v]\{x/u\}$

Regra $(\lambda_y.t1)[z/t2] \rightarrow_{Lamb} \lambda_y.t1[z/t2]$, que pela IH $t \rightarrow t'$
$(\lambda_y.t1)[z/t2]\{x/u\} \rightarrow_{Lamb} \lambda_y.t1[z/t2]\{x/u\}$

Regra $t[z/s][y/v] \rightarrow_{Comp} t[y/v][z/s[y/v]]$,onde $y \in FV(s)$, que pela IH $t \rightarrow t'$
$t[z/s][y/v]\{x/u\} \rightarrow_{Comp} t[y/v][z/s[y/v]]\{x/u\}$

Regra $t[z/s][y/v] \rightarrow t[y/v][z/s]$,onde $y \notin FV(s)$ e $z \notin FV(v)$, que pela IH $t \rightarrow t'$
$t[z/s][y/v]\{x/u\} \rightarrow_C t[y/v][z/s]\{x/u\}$ □

A composição completa nos mostra que substituição explícita implementa substituição implícita, exemplificando melhor, a forma $t[x/u]$ pode ser reduzida para $t\{x/u\}$.

É bem conhecido que confluência nos metatermos falha para o cálculo sem composição para substituição explícita, como por exemplo, o seguinte par crítico no cálculo λ_x mostra abaixo :

$$s = t[x/u][y/v]\ ^* \leftarrow ((\lambda_x.t)\ u)[y/v] \rightarrow^* t[y/v][x/u[y/v]] = s'$$

Este diagrama pode ser fechado no λ_x para termos sem metavariáveis, no entanto não há uma forma de encontrar uma redução comum entre s e s' sempre que t é ou contém metavariáveis, ou seja, não há uma regra de redução λ_x que seja capaz de representar composição nas metavariáveis. Mas, conseguimos fechar este diagrama no cálculo λ_{ex} da seguinte forma : Se $y \in FV(u)$ então $s \rightarrow_{Comp} s'$, da outra forma $s' \rightarrow^*_{ex} (Lema\ 3.6)\ t[y/v][x/u\{y/v\}] = (Lema\ 3.3)\ t[y/v][x/u] =_C s$.

Lema 3.6. *[27](Composição Completa para Metatermos). Sejam t, u metatermos. Então $t[x/u] \rightarrow^*_{ex} t\{x/u\}$*

.

Prova. A prova pode ser feita por indução em t usando $Lema\ 3.3$. Em contraste com a composição completa nos termos, a propriedade mantém-se com uma igualdade para o caso base $t = X_\Delta$ com $x \in \Delta$ desde que $X_\Delta[x/u] = X_\Delta\{x/u\}$

- $t = z$. (t é uma variável)
  A prova se divide em dois subcasos :
  1. $z = x$. $x[x/u] \to_{Var} u = x\{x/u\}$.

  2. $z \neq x$. $z[x/u] \to_{Gc} z = z\{x/u\}$.

- $t = Z_\Delta$. (t é uma meta-variável)
  $Z_\Delta[x/u] =_{def} Z_\Delta\{x/u\}$ se $x \in \Delta$. Se $x \notin \Delta$, então :
  $Z_\Delta[x/u] =_{Gc} Z_\Delta = Z_\Delta\{x/u\}$.

- $t = (t1\ \ t2)$. Seja x uma variável.Temos :
  $(t1\ \ t2)[x/u] \to_{App} t1[x/u]\ \ t2[x/u]$.
  Por hipótese de indução, temos que :
  $t1[x/u] \to^*_{ex} t1\{x/u\}$.
  $t2[x/u] \to^*_{ex} t2\{x/u\}$.
  Então : $(t1\ t2)\{x/u\}$.

- $t = s[y/v]$.
  $s[y/v][x/u]$. Casos a considerar :
  1. $x \in FV(v)$.
  $s[y/v][x/u] \to_{Comp} s[x/u][y/v[x/u]] \overset{IH}{\to}^*_{ex} s\{x/u\}[y/v\{x/u\}]$
  $= s[y/v]\{x/u\} = t\{x/u\}$.

  2. $x \notin FV(v)$ *e* $y \notin FV(u)$.
  $s[y/v][x/u] = c\ s[x/u][y/v] \overset{IH}{\to}^*_{ex} s\{x/u\}[y/v] = s\{x/u\}[y/v\{x/u\}]$
  $= s[y/v]\{x/u\}$. Agora o caso em que :

  $x \notin FV(v)$ *e* $y \in FV(u)$.
  $s[y/v][x/u] = s\{y/y'\}[y'/v][x/u] =_c$
  $s\{y/y'\}[x/u][y'/v] =_{IH} s\{y/y'\}\{x/u\}[y'/v] = s\{y/y'\}\{x/u\}[y'/v\{x/u\}] =_{def}$
  $s\{y/y'\}[y'/v]\{x/u\} =_\alpha s[y/v]\{x/u\}$, onde a equação C pode ser aplicada, pois $y' \notin FV(u)$ *e* $x \notin FV(v)$

- $t = (\lambda_y.t').\quad y \notin FV(u)$:

  $(\lambda_y.t')[x/u] \rightarrow_{Lamb} \lambda_y.t'[x/u] \overset{IH}{\rightarrow}^*_{ex} \lambda_y.t'\,\{x/u\} = (\lambda_y.t')\,\{x/u\}$

- $t = (\lambda_y.t')\,.\quad y \in FV(u)$:

  $(\lambda_y.t')[x/u] \rightarrow_{Lamb} \lambda_z.t'\{y/z\}[x/u] \overset{IH}{\rightarrow}^*_{ex} \lambda_z.t'\{y/z\}\{x/u\} =_{\alpha} \lambda_y.t'\{x/u\} =_{def} (\lambda_y.t')\{x/u\}$

□

O desenvolvimento da prova da confluência para metatermos é baseada na existência de uma função seguindo a verificação da propriedade-Z, como atestado em [34] .

Definição 3.7 (Propriedade-Z)**.** *Uma função $^\circ$ de termos para termos satisfaz a propriedade-Z para uma relação de redução $\rightarrow_R$ se e somente se $t \rightarrow_R u$ implica $u \rightarrow^*_R t^\circ$ e $t^\circ \rightarrow^*_R u^\circ$. Uma relação de redução $\rightarrow_R$ tem a propriedade-Z se existe uma função que satisfaz a propriedade-Z para $\rightarrow_R$.*

De acordo com [34], temos que $\rightarrow_R$ é confluente se ela tem a propriedade-Z (ver Teorema 3.16). Então para mostrar a confluência de λ_{ex} é suficiente definir uma função nos metatermos satisfazendo a propriedade-Z. Tal função pode ser definida em termos da função superdesenvolvimento para o λ cálculo [2, 38] .

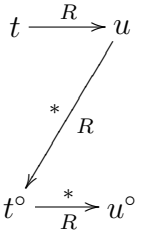


Figura 3.2.1. Propriedade Z de $\rightarrow_R$

Definição 3.8 (Função Superdesenvolvimento)**.** *A função $\_^\circ$ nos metatermos é definida por indução como segue :*

$X^\circ_\Delta := X_\Delta \qquad (tu)^\circ := t^\circ u^\circ \qquad$ *se t° não é uma abstração*

$x^\circ := x \qquad (tu)^\circ := v\{x/u^\circ\} \qquad$ *se* $t^\circ = \lambda_x.v$

$(\lambda_x.t)^\circ := \lambda_x.t^\circ \qquad t[x/u]^\circ := t^\circ\{x/u^\circ\}$

Note que $FV(t^\circ) \subseteq FV(t)$.

As provas e definições apresentadas nesta seção estão formalizadas no capítulo 6. Além disso, no Apêndice A apresentamos a prova da confluência em detalhes [27].

Capítulo 4

Assistente de Prova *Coq*

O *Coq* é um assistente de prova baseado em uma lógica de ordem superior, conhecida como cálculo de construções indutivas, que é uma variação do λ-cálculo tipado [8]. Portanto, utiliza uma teoria de tipos muito expressiva, onde, pelo isomorfismo de Curry-Howard, os tipos são vistos como proposições e os termos como provas [7]. Como citamos na introdução, a linguagem de especificação usada é chamada Ga-llina e é com ela que se definem expressões matemáticas e se realizam as provas. Já a linguagem de comandos é chamada Vernacular. É com ela que indicamos o que iremos fazer, seja iniciar uma definição, iniciar um lema, começar a prova de um lema, etc. [14].

O *Coq* possui um conjunto relativamente extenso de provas nas bibliotecas nativas , além de contribuições de pesquisadores e professores de várias instituições [14]. Isso permite uma interação entre o desenvolvimento de provas o que é importante para o reuso [18].

Todas as expressões em *Coq* possuem um tipo. Se combinarmos expressões, utilizando ou não certos conectivos, para gerar novas expressões bem formadas, estas também terão algum tipo. Por exemplo, declaremos a variável b do tipo nat:

```
Variable b:nat.
```

Podemos agora utilizar a constante 2 do tipo nat para escrever a expressão 2b, que também será do tipo nat. Por outro lado, se utilizarmos a constante true, do tipo bool, ao escrever a expressão true b, haverá um erro, pois não é uma expressão bem formada [8].

Inicialmente, ao carregarmos o *Coq*, temos alguns tipos básicos como nat e $bool$. O tipo de um tipo é chamado de sort. O *Coq* possui 3 sorts pré-definidos:

Prop - para as expressões proposicionais (expressões que podem ser avaliadas como falsas ou verdadeiras, de acordo com sua carga semântica).

Set - para descrever tipos de dados e especificações. Os termos cujo tipo é uma especificação são chamados de programas.

Type - que são utilizados na construção dos universos que estão relacionados à consistência do sistema de tipos. Os identificadores Prop e Set possuem tipo Type, que por sua vez possui tipo Type.

A partir desses três sorts (que também são tipos, já que termos e tipos pertencem à mesma classe no cálculo de construções indutivas), podemos construir

todos os demais tipos que precisarmos. Mas seria muito trabalhoso começarmos apenas com esses tipos para provarmos expressões matemáticas. Justamente tendo em vista o suporte ao reuso, existem os tipos nat, bool, e mais tantos outros que podem ser carregados ao chamarmos as bibliotecas apropriadas (Z, para os inteiros, Q, para os racionais etc).

Há um certo paralelismo entre a teoria de tipos e a teoria de conjuntos [7]. Por exemplo, podemos considerar que elementos de um mesmo tipo pertencem a um mesmo conjunto, o conjunto de todos os elementos com aquele tipo. Vejamos como é construído o tipo nat, que representa o conjunto dos números naturais. Podemos fazer isso em *Coq* com o comando:

```
Print nat.
```

A resposta, será:

```
Inductive nat : Set := O : nat | S : nat -> nat

For S: Argument scope is [nat scope]
```

Isso quer dizer que o tipo nat é do tipo Set (como esperávamos), construído de forma indutiva, com o uso de dois construtores, O, do tipo nat e S, do tipo $nat \rightarrow nat$. O construtor O é do tipo nat. O construtor S requer uma entrada nat para dar uma saída nat ($nat \rightarrow nat$), ou seja, é uma função unária. Assim, S O será do tipo nat. Então, também podemos escrever S (S O), S (S (S O)) etc. Se O é o zero e S é a função sucessor, podemos representar todos os naturais dessa forma. O é o natural 0, S O é o natural 1 (sucessor de 0), S (S O) é 2 etc. E a correspondência de notação é feita no escopo nat scope. Os comandos Check 5 e Check S (S (S (S O)))) geram a mesma saída:

```
5

: nat.
```

Por outro lado, se usarmos o comando Unset Printing Notations, estaremos descarregando as notações do escopo e, agora, ao usarmos o comando Check 5, obteremos a saída:

```
S (S (S (S (S O))))

: nat.
```

No *Coq*, as provas e as especificações permanecem no mesmo arquivo. Após enunciada a especificação, abre-se o campo de provas e podemos começar a aplicar as táticas. Isso é feito de forma interativa. Cada tática é um comando que, inclusive, pode ser desfeito. Existe uma ferramenta gráfica, chamada *CoqIDE* , que auxilia no processo de construção das provas. Também é possível utilizar outras ferramentas de interação, como o Proof General [14].

Podemos enunciar um lema da seguinte forma:

```
Lemma dois_mais_dois: 2+2=4.
```

Isso significa que queremos provar 2+2=4. O identificador dois_mais_dois é utilizado para referências futuras.

Ao iniciarmos a prova, aparecerá o seguinte:

```
1 subgoal

----------------------------(1/1)

2 + 2 = 4
```

Abaixo da linha pontilhada temos um objetivo a ser provado. Acima da linha são colocadas as hipóteses (inexistentes no nosso exemplo). Devido ao isomorfismo de Curry-Howard, uma prova em *Coq* nada mais é do que um termo t da linguagem do cálculo de construções indutivas que possui como tipo o que se está afirmando, no caso 2+2=4. De uma forma geral, a construção de um tal termo não é nada trivial. O *Coq* possui, então, várias táticas que nos permitem construir pouco a pouco esse termo, ou seja, uma prova de 2+2=4. Vamos começar com a tática simpl.

Agora, o campo de prova será

```
1 subgoal

----------------------------(1/1)

4 = 4
```

A tática simpl é uma tática de redução que procura simplificar o termo de prova. Neste caso, ela realizou a soma 2+2. Agora, a prova é trivial, pois 4=4. Podemos usar a tática trivial, que é capaz de identificar igualdades triviais.

```
trivial.
```

E obtemos no campo de provas Proof completed, o que mostra que chegamos ao fim da prova. Ainda precisamos pedir para o *Coq* guardar nossa prova, com o nome que escolhemos no início do comando Lemma, para podermos utilizá-lo futuramente. Só assim sairemos do campo de provas. O comando é

```
Qed.
```

O *Coq* nos dá a saída:

```
simpl in |- *.

trivial.

dois_mais_dois is defined
```

E isso significa que dois_mais_dois foi definido utilizando a tática simpl seguida de trivial. O termo que possui tipo 2+2=4 pode ser visto com o comando Print:

```
Print dois_mais_dois.

refl_equal 4

: 4 = 4
```

Na verdade, esta é uma prova muito simples e trivial. Poderíamos ter simplesmente usado a tática auto, que realiza uma busca completa por definições e provas nas bibliotecas carregadas para tentar provar o objetivo, já que, quando inicialmente carregado, o *Coq* importa parte da biblioteca Arith, que possui, além do tipo nat e do escopo nat_scope, provas envolvendo o tipo nat. Um exemplo bem interessante é a função recursiva abaixo utilizando indução sobre os naturais $(1 + 2 + 3 + ... + n)$:

$Fixpoint\ Sum\ (n:\ nat)\ :\ nat\ :=$
$match\ n\ with$
$\ |\ O\ =>\ O$
$\ |\ (S\ n)\ =>\ (plus\ (S\ n))\ (Sum\ n)$
$end.$
$Sum\ is\ recursively\ defined$

A prova abaixo mostra a utilização da função acima no seguinte lema :

$Lemma\ p\_sum\ :\ forall\ (n:nat),\ 2*(Sum\ n)\ =\ n\ *\ (S\ n).$
$Proof.$
$induction\ n.$
$simpl;\ trivial.$
$simpl\ (Sum\ (S\ n)).$
$simpl\ (S\ n\ *\ S\ (S\ n)).$
$ring\_simplify.$
$rewrite\ IHn.$
$ring.$
$Qed.$

Coq nos possibilita a construção interativa de provas formais, e também a manipulação de programas funcionais consistente com as suas especificações. Ele é executado como um programa de computador em muitas arquiteturas e está disponível com uma variedade de interfaces de usuário [22].

Dando continuidade ao tratamento de tipos em *Coq*, é interessante observar um framework simples fornecido pelo λ-cálculo simplesmente tipado sem polimorfismo, um modelo de linguagens de programação com poder de expressividade reduzido, em que esses tipos têm duas formas :

1. Tipos Atômicos, feitos de identificadores únicos, como nat, Z, e $bool$.

2. Tipos da forma $(A \rightarrow B)$, onde A e B são tipos deles próprios. Por hora, chamaremos esses tipos de tipos $arrow$, que representam tipos de funções: $(A \rightarrow B)$ é o tipo de uma função que toma um objeto do tipo A como argumento e retorna um objeto do tipo B.

Outro ponto interessante, é que podemos introduzir novas definições, que ligam um nome a um valor bem tipado. Por exemplo, podemos introduzir uma constante como sendo definida para ser igual ao sucessor de zero:

$Coq < Definition\ one := (S\ O).$
$one\ is\ defined$

Nós podemos opcionalmente indicar o tipo requerido :

$Coq < Definition\ two : nat := S\ one.$
$two\ is\ defined$

Atualmente *Coq* permite várias possibilidades de sintaxes :

$Coq < Definition\ three : nat := S\ two.$
$three\ is\ defined$

Coq implementa polimorfismo, onde consideramos uma funcionalidade que podemos usar para iterar uma função unária de um tipo a ele mesmo [8]. Por hora, consideramos apenas seu tipo :
$iterate : forall\ A : Set, (A \rightarrow A) \rightarrow nat \rightarrow A \rightarrow A.$

Esta função toma quatro argumentos :

1. Um tipo A,
2. Uma função do tipo $A \rightarrow A$,
3. O número de iteração no tipo nat,
4. O valor inicial do tipo A, no qual a função é iterada o número certo de vezes.

$Check\ (iterate\ nat).$
$iterate\ nat : (nat \rightarrow nat) \rightarrow nat \rightarrow nat \rightarrow nat$

$Check\ (iterate\ \_\ (mult\ 2)).$
$iterate\ nat\ (mult\ 2) : nat \rightarrow nat \rightarrow nat$

$Check\ (iterate\ \_\ (mult\ 2)\ 10).$
$iterate\ nat\ (mult\ 2)\ 10 : nat \rightarrow nat$

$Check\ (iterate\ \_\ (mult\ 2)\ 10\ 1).$

$iterate\ nat\ (mult\ 2)\ 10\ 1\ :\ nat$

$Eval\ compute\ in\ (iterate\ \_\ (mult\ 2)\ 10\ 1).$
$=\ 1024\ :\ nat$

Observa-se que a abordagem de *Coq* é mais simples, no entanto, mais abstrata : tipos podem ser argumentos para as funções da mesma forma que inteiros, listas e funções [8].

$Fixpoint\ iterate\ (A : Set)(f : A \to A)(n : nat)(x : A)\{struct\ n\}\ :\ A\ :=$
$match\ n\ with$
$|\ 0\ \Rightarrow\ x$
$|\ S\ p\ \Rightarrow\ f\ (iterate\ A\ f\ p\ x)$
$end.$

Esta função é interessante porque mostra que uma função recursiva pode tomar uma outra função como argumento. O comando $Fixpoint$ torna possível a programação funcional recursiva de ordem superior. Esta função pode ser comparada a um for, utilizado na programação imperativa, no qual a computação é repetida um dado número de vezes [8].

Um outro tratamento sofisticado de *Coq* é na construção de funções parciais, no qual uma função parcial de tipo A para tipo B pode ser descrita com um tipo da forma :

$forall\ x : A,\ P\ x \to B$, onde P é um predicado que descreve o domínio da função. Aplicar uma função desse tipo requer dois argumentos : um termo t do tipo A e uma prova da pré-condição $P\ t$. Por exemplo :

$Definition\ pred\_partial\ :\ forall\ n\ : nat,\ n \neq 0 \to nat.$, que descreve uma função parcial que somente pode ser usada para números naturais diferentes de zero [8].

Introduzimos agora, o tratamento e construção em *Coq* dos tipos indutivos, por exemplo :
$Inductive\ month\ :\ Set\ :=$
$January\ :\ month\ |\ February\ :\ month\ |\ March\ :\ month$
$|\ April\ :\ month\ |\ May\ :\ month\ |\ June\ :\ month$
$|\ July\ :\ month\ |\ August\ :\ month\ |\ September\ :\ month$
$|\ October\ :\ month\ |\ November\ :\ month\ |\ December\ :\ month.$

Esta definição, simultâneamente introduz um tipo $month$ no sort Set e 12 elementos neste tipo : $January, February$, ... Esses elementos são chamados construtores do tipo indutivo. O sistema *Coq* automaticamente adiciona vários teoremas e funções que tornam possível raciocinar e computar dados desse tipo. O primeiro teorema é chamado $month\_ind$, onde devemos também chamar o princípio indutivo associado com a definição indutiva [8] :

$Check\ month\_ind.$

$month\_ind\ :$
$forall\ P : month \rightarrow Prop,$
$P\ January \rightarrow P\ February \rightarrow P\ March \rightarrow P\ April \rightarrow$
$P\ May \rightarrow P\ June \rightarrow P\ July \rightarrow P\ August \rightarrow P\ September \rightarrow$
$P\ October \rightarrow P\ November \rightarrow P\ December \rightarrow\ forall\ m : month,\ P\ m$

A conclusão indica que a propriedade P mantém-se para todos $months$. Similarmente *Coq* também gera uma função chamada $month\_rec$ e $month\_rect$, onde no primeiro caso a quantificação inicial manipula uma propriedade cujo valor é um Set, e no segundo caso, o valor é $Type$.

Coq pode ser utilizado para realizar extração de código, uma importante qualidade da ferramenta, onde usa uma tradução de programas funcionais dentro de programas funcionais em uma linguagem de programação eficiente, como OCAML ou Haskell, ou melhor, realiza o mapeamento de cada função de um desenvolvimento formal a correspondente função na linguagem de programação, e este mapeamento é chamado de extração [8].

Extração em *Coq* é, portanto, um rico framework que permite obter programas certificados expressos em OCaml, Haskell ou Scheme, essas três linguagens tem sido atualmente suportadas pela extração em *Coq* [30]. A principal motivação para o mecanismo de extração é produzir programas certificados : cada propriedade comprovada em Coq ainda será válida após a extração. [30]. Ilustramos abaixo, um exemplo de uso de extração em Coq : divisão euclidiana entre números naturais.

$Fixpoint\ div\ x\ y\ :=\ match\ x\ with$
$|\ 0\ =>\ 0$
$|\ S\ x'\ =>$
$\quad let\ z\ :=\ div\ x'\ y\ in$
$\quad if\ le\_lt\_dec\ ((S\ z) * y)\ x\ then\ S\ z\ else\ z$
$end.$

O comando $Extraction\ div$ pode então ser usado para converter esta divisão para código Ocaml :

$let\ rec\ div\ x\ y\ =$
$\quad match\ x\ with$
$\quad\quad |\ O\ \rightarrow\ O$
$\quad\quad |\ S\ x'\ \rightarrow$
$\quad\quad\ let\ z\ =\ div\ x'\ y\ in$
$\quad\quad\ if\ le\_lt\_dec\ (mult\ (S\ z)\ y)\ x\ then\ S\ z\ else\ z$

Capítulo 5

Lógica Nominal

O principal objetivo dessa abordagem é permitir a especificação da α-conversão, que evita explicitamente lidar com a captura de variáveis, onde podemos verificar melhor na seguinte operação $s\{x/v\}$ definida no módulo α-conversão por indução em s como abaixo :

$$
\begin{array}{lll}
x\{x/v\} = v & & \\
y\{x/v\} = y & se & x \neq y \\
(tu)\{x/v\} = (t\{x/v\}u\{x/v\}) & & \\
(\lambda y.t)\{x/v\} = \lambda y.(t\{x/v\}) & se & x \neq y \\
t[y/u]\{x/v\} = t\{x/v\}[y/u\{x/v\}] & &
\end{array}
$$

Exemplificando,

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

Lógica Nominal é um formalismo para teorias que envolvem ligações (*binders*) via permutação de nomes, que permite expressar as propriedades lógicas por meio da noção de renomeamento de átomos. Ao mesmo tempo, visa proporcionar um fundamento suficiente para a teoria de indução e recursão estrutural para sintaxe módulo α-conversão [35].

Apenas para a lógica de primeira ordem, uma teoria em lógica nominal é especificada por uma assinatura contendo *sorts*, funções e símbolos de relação, juntos com uma coleção de axiomas, os quais são fórmulas de primeira ordem construídas na forma usual a partir de variáveis e os símbolos da assinatura, mas agora, usando as funções permutação e uma relação para variáveis novas (*freshness*).

Neste trabalho, está um exemplo de como esta linguagem de lógica nominal pode ser usada para formalizamos alguns conceitos familiares do lambda cálculo dentro do assistente de prova *Coq*. Essa abordagem é caracterizada pela correspondência entre a prova realizada no papel e raciocinando (pensando logicamente) dentro de *Coq*. Por exemplo :

- Todas as ocorrências de variáveis de um dado sort (livres ou ligadas), são representadas uniformemente usando átomos, um conjunto infinito de objetos com

igualdade decidível [3].

- α-equivalência de termos é representada pela igualdade pré-construída do *Coq* e não uma relação de equivalência definida separadamente [3].

Ambos os pontos acima, refletem a prática comum com formalizações em papel e lápis. Mas genericamente, essa abordagem nominal é projetada para eliminar a necessidade de pensar sobre alguns termos que não aperecem atualmente nas provas em papel, isto é, pré-termos, shifted termos, e termos exóticos.

Alguns fundamentos sobre lógica nominal são apresentados. Então, começamos descrevendo um tipo de conjunto finito com igualdade extensional usando o construtor record , parte do qual é mostrado abaixo :

$$Record\ ExtFset\ (T\ :\ Set)\ :\ Type\ :=\ mkExtFset\ \{$$
$$extFset\ :\ Set;\ In\ :\ T \rightarrow extFset \rightarrow Prop;\ ...\ \}$$

O tipo record é parametrizado por T, o tipo dos elementos carregados pelo conjunto. O tipo atual dos conjuntos finitos sobre T é dado pelo campo $extFset$, e In é um predicado de um membro relacionado com o conjunto. O nome desses campos são constantes, cujos tipos completos são :

$$extFset\ :\ \forall\ T\ :\ Set,\ ExtFset\ T \rightarrow Set$$
$$In\ :\ \forall\ (T\ :\ Set)\ (R\ :\ ExtFset\ T),\ T \rightarrow extFset\ T\ R \rightarrow Prop.$$

Nós usamos o mecanismo de argumentos implícitos de *Coq* para inferir os argumentos T e R quando possível, e escrevemos $x \notin F$ para $not\ (In\ x\ F)$ quando isso pode ser feito. Para melhor ilustrar o uso desses conjuntos, observe uma definição utilizada na especificação do cálculo λ_{ex} :

$$Record\ ExtFset\ (A\ :\ Set)\ :\ Type\ :=\ mkExtFset\ \{$$
$$extFset\ :\ Set;$$
$$(*\ Membros.\ *)$$
$$In\ :\ A\ ->\ extFset\ ->\ Prop;$$
$$In\_dec\ :\ forall\ x\ E,\ \{In\ x\ E\}\ +\ \{\tilde{}\ In\ x\ E\};$$

$$(*\ Igualdade.\ *)$$
$$eq\_carrier\_dec\ :\ forall\ (x\ y\ :\ A),\ \{x\ =\ y\}\ +\ \{x\ <>\ y\};$$
$$eq\_extFset\ :\ forall\ E\ F,\ (forall\ x,\ In\ x\ E\ <->\ In\ x\ F)\ ->\ E\ =\ F;$$

$$(*\ Intersection.\ *)$$
$$intersection\ :\ extFset\ ->\ extFset\ ->\ extFset;$$
$$intersection\_intro\ :\ forall\ xE\ F,\ In\ x\ E\ ->\ In\ x\ F\ ->\ In\ x\ (intersection\ E\ F);$$
$$intersection\_elim\_1\ :\ forall\ x\ E\ F,\ In\ x\ (intersection\ E\ F)\ ->\ In\ x\ E;$$
$$intersection\_elim\_2\ :\ forall\ x\ E\ F,\ In\ x\ (intersection\ E\ F)\ ->\ In\ x\ F;$$
$$\}$$

Após a idéia de conjuntos, é interessante observar o uso do records, que implementa um dicionário semântico para classes type, e ao mesmo tempo nos mostra a definição de átomos :

$$
\begin{array}{l}
Record \quad AtomT : Type := mkAtom\{ \\
atom : Set; \quad asetR : ExtFset \quad atom; aset := extFset \quad asetR; \\
atom\_eqdec : \forall\ a \quad b : atom, \{a = b\} + \{a <> b\}; \\
atom\_infinite : \forall\ F : aset, \{b : atom \quad | \quad b \notin F\}\}.
\end{array}
$$

Figura 5.1. Átomos

$$
\begin{array}{l}
Record \quad SwapT(A : AtomT)(X : Set) : Set := mkSwap\{ \\
swap : (A * A) \rightarrow X \rightarrow X; \\
swap\_same : \forall\ a \quad x, swap\ (a, a)\ x = x; \\
swap\_invol : \forall\ a \quad b \quad x, swap\ (a,\ b) \quad (swap\ (a,\ b)\ x) = x; \\
swap\_distrib : \forall\ a \quad b \quad c \quad d \quad x, \\
swap\ (a,\ b)\ (swap\ (c,\ d)\ x) = \\
swap \quad (swapa \quad A \quad (a,\ b)\ c, \quad swapa \quad A \quad (a,\ b)\ d) \quad (swap\ (a,\ b)\ x)\}.
\end{array}
$$

Figura 5.2. Permutação

Cada tipo record define uma classe type, e campos do tipo record são campos da classe type. Usa-se um tipo record para capturar qualidades essenciais dos nomes das variáveis em nossa linguagem de objetos, sabe-se que existem um número infinito de nomes e que igualdade em nomes é decidível. Chamamos objetos com essas propriedades de átomos; records do tipo $AtomT$, como mostrado na Figura 5.1, consiste de um tipo e provas que o type é uma coleção de átomos. O campo $atom$ é o tipo de átomos, $aset$ é o tipo de conjuntos finito de átomos, e $atom\_eqdec$ assume que igualdade nos átomos é decidível.

A função $atom\_infinite$, quando fornecido algum conjunto finito F de átomos, produz um átomo b juntamente com uma prova de que b não está em F. Note que essa função requer que o tipo atom seja infinito e implemente um átomo novo, uma operação cujos detalhes são tipicamente deixados, não especificados em papel. Com as coerções implícitas de *Coq*, para algum $A \ : \ AtomT$, podemos escrever A onde quer que átomo A seja requisitado. Especificamente, sempre que A ocorre em uma localização onde um termo do tipo Set é requisitado, *Coq* implicitamente insere uma aplicação do átomo.

Tendo caracterizado átomos, precisa-se construir uma definição para *swapping* de um par de átomos em expressões arbitrárias. *Swapping* de átomos é um conceito central em abordagens nominal por duas razões. Primeiro é fácil de definir um método apropriado de *swapping* de átomos em quase todo tipo, incluindo tipos funcionais e tipos com *nominal binding*. Segundo, dá-nos um sentido genericamente especificado cujos nomes são novos para algum tipo. Importantes propriedades de permutação de átomos para algum tipo X são especificadas pelo record $SwapT$ na Figura 5.2. A propriedade $swap\_same$ assume que *swapping* de um átomo com ele mesmo deve sempre levar a expressão não modificada. A próxima propriedade

atesta que *swapping* deve ser uma involução. A propriedade final permite aninhar swaps para serem reordenados. Na teoria, o usuário pode utilizar alguma definição de swapping para um dado tipo que satisfaz as propriedades em $SwapT$, mas na prática existe usualmente um natural definido pela estrutura de type [3]. A forma mais simples de *swapping* é o swap de átomos a e b de tipo $atom\ A$ aplicado ao $atom\ C$, também do tipo $atom\ A$, denotado por $swapa\ A\ (a,\ b)\ c$. O construtor $mkAtomSwap$ usa a função $swapa$ para construir o record $SwapT$. Para tipos onde não aparece átomos(o tipo nat), a única definição razoável de aplicar um swap é deixar o objeto inalterado.

Definir como aplicar um swap numa expressão com um tipo funcional não é tão complicado assim. Nossa definição segue [36] e satisfaz as propriedades no record $SwapT$(Se nos permite aplicar um axioma de extensionalidade funcional) :

$$Variables\quad (A : AtomT).$$
$$Variables\quad (X : Set)(XS : SwapT\quad A\quad X)(Y : Set)(YS : SwapT\quad A\quad Y).$$
$$Definition\quad func\_swap\quad (a\quad b : A)(f : X \to Y) :=$$
$$fun\quad x \Rightarrow swap\quad YS\quad (a, b)\quad (f\quad (swap\quad XS\quad (a, b)\quad x)).$$

Observe que $atom\ swapping$ permite aos usuários definir swapping em algum tipo não dependente que habita no sort Set.

Um outro ponto de suma importância nessa abordagem é a definição de suporte finito, cuja idéia é a seguinte : a função fornece que, cada átomo ligado, deve satisfazer uma condição nova para *binders* (FCB) dizendo, que para alguma escolha suficientemente nova do átomo ligado, o resultado da função nunca pode conter que o átomo está dentro do seu suporte. Essas condições asseguram que existe uma única família de funções suportada finitamente de aridade-indexada que está bem definida na classe de α-equivalência [36].

$$Variables\ (A\ :\ AtomT)\ (X\ :\ Set)\ (S\ :\ SwapT\ A\ X).$$
$$Definition\ supports\ (F\ :\ aset\ A)\ (x\ :\ X)\ :\ Prop\ :=$$
$$\forall\ a\ b\ :\ A,\ a \notin F -> b \notin F -> swap\ S\ (a,\ b)\ x\ =\ x.$$
$$Definition\ fresh\ (b\ :\ A)\ (x\ :\ X)\ :\ Prop\ :=$$
$$\exists\ F\ :\ aset\ A,\ supports\ F\ x \wedge b \notin F.$$

Maiores informações podem ser encontradas em [3].

É interessante observar e compreender bem os conceitos relativos a átomos e swap, que serão bastante utilizados na especificação do cálculo λ_{ex} no próximo capítulo. Todas as ocorrências de variáveis de um dado *sort* são representadas uniformemente usando átomos, um conjunto infinito de objetos com igualdade decidível. Uma vez caracterizado átomos, precisa-se construir uma definição para *swapping*, como um par de átomos em expressões arbitrárias. *Swapping* de átomos é um conceito central em abordagens nominais, haja vista que é fácil definir um método apropriado de swapping de átomos em praticamente qualquer tipo, e dá-nos um meio para especificar quais nomes são novos para qualquer tipo. Apresenta-se

abaixo essa relação :

$$
\begin{array}{l}
Definition\ swapa\ (s\ :\ atom\ A\ *\ atom\ A)\ (c\ :\ atom\ A)\ :=\\
let\ (a,\ b)\ :=\ s\ in\\
if\ atom\_eqdec\ \_\ a\ c\ then\ b\\
else\ if\ atom\_eqdec\ \_\ b\ c\ then\ a\\
else\ c.
\end{array}
$$

Definição de swaps para termos:

$$
\begin{array}{l}
swap :=\\
\left\{\begin{array}{ll}
swap\_same: & \forall a\, x, swap\ \ (a,a)\ x = x\\
swap\_invol: & \forall a\, b\, x, swap\ \ (a,b)(swap\ \ (a,b)\ x) = x\\
swap\_distrib: & \forall a\, b\, c\, d\, x, swap\ \ (a,b)(swap\ \ (c,d)\ x) =\\
 & swap\ \ (swapa\ \ (a,b)\ c, swapa\ \ (a,b)\ d)(swap\ \ (a,b)\ x)
\end{array}\right.
\end{array}
$$

- Permutações são listas de swaps. Ex.: $[(a,b),(c,d),(a,e)]$

Todos esses conceitos e definições são utilizados na especificação do cálculo λ_{ex} apresentada no próximo capítulo. É importante notar que a principal vantagem da utilização da abordagem nominal é permitir uma apresentação bem próxima à realizada em papel e lápis.

Capítulo 6

Especificação do Cálculo λ_{ex} em *Coq*

As definições vistas na seção sobre o cálculo λ_{ex} estão implementadas em *Coq* seguindo a representação abaixo :

Inicialmente, começamos com uma definição da gramática do λex:

$Variable\ tmvar\ :\ Set.$

$Inductive\ tm\ :\ Set\ :=$
$|\ var\ :\ tmvar \rightarrow tm$
$|\ app\ :\ tm \rightarrow tm \rightarrow tm$
$|\ abs\ :\ tmvar \rightarrow tm \rightarrow tm$
$|\ subs\ :\ tm \rightarrow tmvar \rightarrow tm \rightarrow tm.$

Para definição do conjunto dos x-termos, foi criado um tipo indutivo $[Phi\ n]$, que representa o conjunto de todos os termos cujas variáveis ligadas tem índice menor do que $[n]$. Assim, $[Phi\ 0]$ corresponde ao conjunto dos termos válidos.

Variáveis livres ainda são representadas usando nomes e substituições explícitas devem ser especificadas utilizando-se nomes para as variáveis, e não índices de deBruijn como é feito para as variáveis ligadas por abstratores.

$Inductive\ Phi\ :\ nat \rightarrow Set\ :=$
$|\ pdot\ :\ forall\ (n\ :\ nat),\ Phi\ n$
$|\ pfree\ :\ forall\ (n\ :\ nat),\ tmvar\ \rightarrow\ Phi\ n$
$|\ pbound\ :\ forall\ (n\ i\ :\ nat),\ i\ <\ n\ \rightarrow\ Phi\ n$
$|\ papp\ :\ forall\ (n\ :\ nat),\ Phi\ n\ \rightarrow Phi\ n\ \rightarrow\ Phi\ n$
$|\ plam\ :\ forall\ (n\ :\ nat), Phi\ (S\ n)\ \rightarrow\ Phi\ n$
$|\ psubst\ :\ forall\ (n\ :\ nat),\ Phi\ (S\ n)\ \rightarrow\ Phi\ n\ \rightarrow\ Phi\ n\ .$

Baseado na construção acima, definimos uma notação de alto nível para os construtores de termo e tipos :

$Definition\ tm\ :\ Set\ :=\ Phi\ 0.$
$Definition\ tmP\ :=\ PhiPermR\ 0.$
$Definition\ dot\ :\ tm\ :=\ pdot\ 0.$
$Definition\ var\ (a\ :\ tmvar)\ :\ tm\ :=\ pfree\ 0\ a.$
$Definition\ app\ (s\ t\ :\ tm)\ :\ tm := papp\ 0\ s\ t.$
$Definition\ lam\ (a\ :\ tmvar)\ (s\ :\ tm)\ :\ tm\ :=\ plam\ 0\ (abs\ a\ s).$
$Definition\ subst\ (s\ :\ tm)\ (a\ :\ tmvar)(t\ :\ tm)\ :\ tm\ :=\ psubst\ 0\ (abs\ a\ s)\ t.$

Pode-se formalizar a α-equivalência para abstrações e substituições explícitas da seguinte maneira :

$Lemma\ eq\_lam\ :\ forall\ a\ b\ s,$
$\tilde{}\ In\ b\ (fvar\ s)\ \rightarrow\ lam\ a\ s\ =\ lam\ b\ (perm\ tmP\ [(a,\ b)]\ s).$

$Lemma\ eq\_subst\ :\ forall\ a\ b\ s\ t,$
$\tilde{}\ In\ b\ (fvar\ s)\ \rightarrow\ subst\ a\ s\ t\ =\ subst\ b\ (perm\ tmP\ [(a,b)]\ s)\ t.$

Observa-se no primeiro lema, que se b não pentence ao conjunto de variáveis livres de s, pode-se realizar a permutação da variável ligada a por b, não havendo perda semântica na construção, que corresponde a noção de α-equivalência. A mesma idéia se aplica ao segundo lema, que permite renomear variáveis ligadas na susbtituição explícita.

O uso da ação de permutação de átomos nos termos nos passa a idéia de uma formalização aparentemente simples, no entanto requer que tenhamos um rigoroso cuidado na sua utilização, pois o emprego da técnica de forma incorreta, em indução e recursão α-estrutural, pode gerar problemas sérios no comportamento dessas construções [37].

A definição da meta-substituição é feita de forma recursiva, como evidenciado a seguir :

$Definition\ meta\_subst\ :\ tm\ \rightarrow\ tm\ :=$
$tm\_rec\ tm$
$\quad dot$
$\quad (fun\ x\ \Rightarrow\ if\ atom\_eqdec\ tmvar\ x\ y\ then\ s\ else\ (var\ x))$
$\quad (fun\ s\ s'\ t\ t'\ \Rightarrow\ app\ s'\ t')$
$\quad (fun\ x\ t\ t'\ \Rightarrow\ lam\ x\ t')$
$\quad (fun\ x\ s\ s'\ t\ t'\ \Rightarrow\ subst\ x\ s'\ t')$
$\quad (add\ y\ (fvar\ s)).$

A manipulação da operação de substituição explícita se dá através de símbolos e regras de redução, dentro da sintaxe do sistema, as quais são implementadas em *Coq* da forma abaixo, estando em consonância com o declarado na Figura 3.2 :

$t[x/u][y/v] =_C t[y/v][x/u]$ *se* $y \notin \mathbf{FV}(u)$ & $x \notin \mathbf{FV}(v)$

$Axiom\ equation\_C\ :\ forall\ (t\ u\ v : tm)\ (x\ y : tmvar),$
$\tilde{}\ In\ y\ (fvar\ u)\ \rightarrow \tilde{}\ In\ x\ (fvar\ v)\ \rightarrow$
$t\ \hat{}[x \mid u]\ \hat{}[y \mid v]\ =\ t\ \hat{}[y \mid v]\ \hat{}[x \mid u].$

Dado três termos $t, u\ e\ v$, e duas variáveis x, y, se y não está no conjunto de variáveis livres de u, e x não está no conjunto de variáveis livres de v, então aplica-se a equação de comutatividade para substituições independentes.

$(\lambda x.s)t \rightarrow_B s[x/t]$

$Inductive\ rule\_b\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_b\ :\ forall\ (s\ t\ :\ tm)\ (x\ :\ tmvar),\ rule\_b\ ((\&\ x,\ s)\ !\ t)\ (s\ \hat{}[x \mid t]).$

Dado dois termos $s\ e\ t$, e uma variável x, aplica-se a regra B, que substitui x por t, na estrutura de s.

$x[x/u] \rightarrow_{Var} u$

$Inductive\ rule\_var\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_var\ :\ forall\ (u\ :\ tm)\ (x\ :\ tmvar),\ rule\_var\ ((var\ x)\ \hat{}[x \mid u])\ u.$

Dado um termo u, e uma variável x, aplica-se a regra Var, que substitui x por u em x, dando como resultado o termo u.

$t[x/u] \rightarrow_{Gc} t$ *se* $x \notin \mathbf{FV}(t)$

$Inductive\ rule\_gc\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_gc\ :\ forall\ (t\ u\ :\ tm)\ (x\ :\ tmvar),\ \tilde{}\ In\ x\ (fvar\ t)\ \rightarrow\ rule\_gc\ (t\ \hat{}[x \mid u])\ t.$

Dado dois termos $t\ e\ u$, e uma variável x, se x não pertence ao conjunto de variáveis livres de t, então aplica-se a regra Gc, dando como resultado o próprio t, ou seja, a substituição não tem efeito.

$(t\ u)[x/v] \rightarrow_{App} t[x/v]\ u[x/v]$

$Inductive\ rule\_app\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_app\ :\ forall\ (t\ u\ v :\ tm)\ (x\ :\ tmvar),$
$rule\_app\ ((t\ !\ u)\ \hat{}[x \mid v])\ ((t\ \hat{}[x \mid v])\ !\ (u\ \hat{}[x \mid v])).$

Dado três termos $t, u\ e\ v$, e uma variável x, aplica-se a regra App. Onde aplica-se a substituição de x por v no primeiro termo t e no segundo termo, u, ou seja, é distribuída.

$(\lambda x.s)[y/t] \rightarrow_{Lamb} \lambda x.s[y/t]$

$Inductive\ rule\_lamb\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_lamb\ :\ forall\ (s\ t\ :\ tm)\ (x\ y\ :\ tmvar),$
$x <> y- >^{\sim} In\ x\ (fvar\ t) \rightarrow\ rule\_lamb\ ((\&\ x,\ s)\ \hat{}[y \mid t])\ (\&\ x,\ (s\ \hat{}[y \mid t])).$

Dado dois termos $s,\ t$ e duas variáveis $x,\ y$. Se x for diferente de y, e x não pertencer ao conjunto de variáveis livres de t, aplica-se a regra $lamb$, ocorrendo apenas a propagação.
$t[x/u][y/v] \rightarrow_{Comp} t[y/v][x/u[y/v]]$ *se* $y \in \mathbf{FV}(u)$

$Inductive\ rule\_comp\ :\ tm \rightarrow tm \rightarrow Prop\ :=$
$reg\_rule\_comp\ :\ forall\ (t\ u\ v\ :\ tm)\ (x\ y\ :\ tmvar),$
$x <> y \rightarrow^{\sim} In\ x\ (fvar\ v) \rightarrow In\ y\ (fvar\ u) \rightarrow$
$rule\_comp\ ((t\ \hat{}[x \mid u])\ \hat{}[y \mid v])\ ((t\ \hat{}[y \mid v])\ \hat{}[x \mid (u\ \hat{}[y \mid v])]).$

Dado três termos $t, u\ e\ v$, e duas variáveis x, y. Se y pertence ao conjunto de variáveis livres de u, aplica-se a regra $Comp$ para substituições dependentes.

Com a especificação das regras, partimos para definição dos sistemas de redução, que são gerados por todas as regras de reescrita vistas acima, onde mostra-se a aplicação das regras nos termos, como apresentado abaixo :

Aplicação das relações de reescrita nos termos, de acordo com o sistema ex, que corresponde a equação C (comutatividade de substituições independentes) e às regras $Var, Gc, App, Lamb$ e $Comp$, onde para todo termo M e N, aplicamos a equação e as regras citadas.

$Inductive\ sistema\_ex\ :\ tm \rightarrow tm \rightarrow\ Prop\ :=$
$\mid rel\_rule\_var\_ex\ :\ forall\ M\ N\ :\ tm,\ (rule\_var\ M\ N) \rightarrow\ sistema\_ex\ M\ N$
$\mid rel\_rule\_gc\_ex\ :\ forall\ M\ N\ :\ tm,\ (rule\_gc\ M\ N) \rightarrow\ sistema\_ex\ M\ N$
$\mid rel\_rule\_app\_ex\ :\ forall\ M\ N\ :\ tm,\ (rule\_app\ M\ N) \rightarrow\ sistema\_ex\ M\ N$
$\mid rel\_rule\_lamb\_ex\ :\ forall\ M\ N\ :\ tm,\ (rule\_lamb\ M\ N) \rightarrow\ sistema\_ex\ M\ N$
$\mid rel\_rule\_comp\_ex\ :\ forall\ M\ N\ :\ tm,\ (rule\_comp\ M\ N) \rightarrow\ sistema\_ex\ M\ N.$

Aplicação das relações de reescrita nos termos, de acordo com o sistema Bex, que corresponde às regras do sistema ex mais a regra $Beta$, onde para todo termo M e N, aplicamos as regras citadas.

$Inductive\ sistema\_Bex\ :\ tm \rightarrow tm \rightarrow\ Prop\ :=$
$\mid rel\_rule\_b\_Bex\ :\ forall\ M\ N\ :\ tm,\ (rule\_b\ M\ N) \rightarrow\ sistema\_Bex\ M\ N$
$\mid rel\_sistema\_ex\ :\ forall\ M\ N\ :\ tm,\ (sistema\_ex\ M\ N) \rightarrow\ sistema\_Bex\ M\ N.$

Além da definição dos sistemas, vistas acima, faz-se necessário a definição dos predicados $[rel\_ex]$ e $[rel\_Bex]$, que correspondem a noção de redução de termos, a

seguir apresentado :

Aplicação, sem perda de generalidade, do sistema ex na raíz do termo, e propagando as reduções nos termos de acordo com o conjunto dos termos definido na gramática do cálculo, como por exemplo,na aplicação, faz-se a redução no termo do lado esquerdo e direito da aplicação, na abstração, faz-se a redução no termo da abstração e na substituição, faz-se a redução nos dois termos da substituição, ou seja, no primeiro termo(fun) e no argumento da substituição(arg).

$Inductive\ rel\_ex\ :\ tm \to tm \to Prop\ :=$
$|\ rel\_root\_ex\ :\ forall\ (a\ b : tm),\ sistema\_ex\ a\ b \to\ rel\_ex\ a\ b$
$|\ rel\_app\_l\_ex\ :\ forall\ (a\ b\ b' : tm),\ rel\_ex\ a\ b \to\ rel\_ex\ (app\ a\ b')\ (app\ b\ b')$
$|\ rel\_app\_r\_ex\ :\ forall\ (a\ b\ b' : tm),\ rel\_ex\ a\ b \to\ rel\_ex\ (app\ b'\ a)\ (app\ b'\ b)$
$|\ rel\_abs\_ex\ :\ forall\ (x : tmvar)\ (a\ b : tm),\ rel\_ex\ a\ b \to\ rel\_ex\ (lam\ x\ a)\ (lam\ x\ b)$
$|\ rel\_subst\_fun\_ex\ :\ forall\ (a\ b\ b' : tm)\ (x : tmvar),\ rel\_ex\ a\ b \to\ rel\_ex\ (subst\ x\ a\ b')$
$(subst\ x\ b\ b')$
$|\ rel\_subst\_arg\_ex\ :\ forall\ (a\ b\ b' : tm)\ (x : tmvar),\ rel\_ex\ a\ b \to\ rel\_ex\ (subst\ x\ b'\ a)$
$(subst\ x\ b'\ b).$

Aplicação, sem perda de generalidade, do sistema Bex na raíz do termo, e propagando as reduções nos termos de acordo com o conjunto dos termos definido na gramática do cálculo, como por exemplo,na aplicação, faz-se a redução no termo do lado esquerdo e direito da aplicação, na abstração, faz-se a redução no termo da abstração e na substituição, faz-se a redução nos dois termos da substituição, ou seja, no primeiro termo(fun) e no argumento da substituição(arg).

$Inductive\ rel\_Bex\ :\ tm \to tm \to Prop\ :=$
$|\ rel\_root\_Bex\ :\ forall\ (a\ b : tm),\ sistema\_Bex\ a\ b \to\ rel\_Bex\ a\ b$
$|\ rel\_app\_l\_Bex\ :\ forall\ (a\ b\ b' : tm),\ rel\_Bex\ a\ b \to\ rel\_Bex\ (app\ a\ b')\ (app\ b\ b')$
$|\ rel\_app\_r\_Bex\ :\ forall\ (a\ b\ b' : tm),\ rel\_Bex\ a\ b \to\ rel\_Bex\ (app\ b'\ a)\ (app\ b'\ b)$
$|\ rel\_abs\_Bex\ :\ forall\ (x : tmvar)\ (a\ b : tm),\ rel\_Bex\ a\ b \to\ rel\_Bex\ (lam\ x\ a)$
$(lam\ x\ b)$
$|\ rel\_subst\_fun\_Bex\ :\ forall\ (a\ b\ b' : tm)\ (x : tmvar),\ rel\_Bex\ a\ b \to\ rel\_ex\ (subst\ x\ a\ b')$
$(subst\ x\ b\ b')$
$|\ rel\_subst\_arg\_Bex\ :\ forall\ (a\ b\ b' : tm)\ (x : tmvar),\ rel\_Bex\ a\ b \to\ rel\_ex\ (subst\ x\ b'\ a)$
$(subst\ x\ b'\ b).$

Redução nos termos em 1 ou mais passos, correspondendo ao fecho transitivo de $[rel\_Bex]$.

$Inductive\ explicit\_rel\_plus$
$(R\ :\ A\ \to\ A\ \to\ Prop)\ :\ A\ \to\ A\ \to\ Prop\ :=$
$|\ relplus\_1step\ :\ forall\ x\ y\ :\ A,\ R\ x\ y\ \to\ explicit\_rel\_plus\ R\ x\ y$
$|\ relplus\_trans1\ :\ forall\ x\ y\ z\ :\ A,\ R\ x\ y\ \to\ explicit\_rel\_plus\ R\ y\ z\ \to$
$explicit\_rel\_plus\ R\ x\ z.$

O fecho reflexivo transitivo da relação $[rel\_Bex]$.

$Inductive\ explicit\_star$
$(R\ :\ A\ \rightarrow\ A\ \rightarrow\ Prop)\ :\ A\ \rightarrow\ A\ \rightarrow\ Prop\ :=$
$|\ star\_refl\ :\ forall\ x\ :\ A,\ explicit\_star\ R\ x\ x$
$|\ star\_trans1\ :\ forall\ x\ y\ z\ :\ A,$
$R\ x\ y\ \rightarrow\ explicit\_star\ R\ y\ z\ \rightarrow\ explicit\_star\ R\ x\ z.$

6.1 Composicionalidade do Cálculo λ_{ex} em *Coq*

Neste ponto, após a especificação do Cálculo λ_{ex} realizada acima, e em consonância com a prova da confluência realizada no capítulo três, apresentaremos a formalização mecânica dessa prova em *Coq*, ou melhor, dos principais resultados alcançados, ressaltando que, devido ser um trabalho não trivial e desafiador, apenas enfatizaremos os Lemas e definições já provados, ficando as provas pendentes para trabalhos futuros. Então, como ponto de partida, iniciamos com o seguinte Lema abaixo :

$Lemma\ meta\_subst\_not\_fv\ :\ forall\ (x\ :\ tmvar)\ (N\ M\ :\ tm),$
$\tilde{}\ In\ x\ (fvar\ M) \rightarrow M\ \hat{}\{x := N\}\ =\ M.$

Refere-se ao primeiro lema apresentado na prova da confluência realizada na subseção 3.2, *Lema* 3.3, dizendo que que se x não pertence ao conjunto de variáveis livres do termo M, então a substituição não tem efeito, dando como resultado o próprio termo M. A prova se dá por indução em M, onde será expandida em cinco casos, considerando os cinco construtores definidos na indução elaborada na especificação do cálculo, e não a indução pré-construída de *Coq*. A prova será concluída quando todos os objetivos gerados em cima dos construtores $dot,\ var,\ app,\ lam$ e $subst$ tiverem sido provados.

A próxima prova diz respeito a composição, *Lema* 3.4 (Composição), mostrando que as duas equações pertencem à mesma classe de equivalência :

$Lemma\ meta\_subst\_composition\ :\ forall\ (x\ y\ :\ tmvar)\ (t\ u\ v\ :\ tm),$
$x <> y \rightarrow \tilde{}\ In\ x\ (fvar\ v) \rightarrow$
$t\ \hat{}\{x := u\}\ \hat{}\{y := v\} = t\ \hat{}\{y := v\}\ \hat{}\{x := u\ \hat{}\{y := v\}\}.$

A prova se dá por indução em t, desenvolvendo-a com a mesma idéia do $Lemma$ $meta\_subst\_not\_fv$, provando os subobjetivos gerados para os construtores mencionados acima, e utilizando o resultado anterior para conclusão da prova.

Neste ponto, já temos material suficiente para apresentar em mais detalhes uma prova não trivial do cálculo-λ_{ex} em *Coq*. Como exemplo, apresentamos o lema da substituição :

```
Lemma meta_subst_composition :  forall (x y :  tmvar)
(t u v :  tm), x <> y →  ~ In x (fvar v) →
t ^{x := u} ^{y := v} = t ^{y := v} ^{x := u ^{y := v}}.

Proof.
```

- Primeiramente introduzimos as hipóteses, ou seja, variáveis que representam proposições no contexto.

  ```
  intros x y t u v xy xv.
  ```

- Em seguida aplica-se a indução construída na especificação, gerando os objetivos a serem provados em função dos construtores *dot*, *var*, *app*, *lam* e *subst*, onde *dot* é um construtor para constantes, *var* para variáveis, *app* para aplicações, *lam* para abstrações e *subst* para a substituição explícita.

  ```
  pattern t; apply tm_induction'.
  ```

- (* *dot* *) Para o caso da constante a prova sai de forma simples, bastando aplicar a tática abaixo, a qual procura dentro das definições da especificação algo que se aplica para este caso.

  ```
  autorewrite with lex; trivial.
  ```

- (* *var* *) No caso de *var*, precisou-se aplicar *atom_eqdec*, um construtor dentro do *Record AtomT*, como apresentado no capítulo cinco, que garante a decidibilidade entre os átomos.

  ```
  intros z.
  ```

  ```
  case (atom_eqdec _ z x).  case (atom_eqdec _ z y).
  ```

  ```
  intros H H'. rewrite H in H'. apply sym_eq in H'.
  ```

  ```
  contradiction.
  ```

  Neste ponto, ainda dentro do construtor *var*, utilizamos os seguintes lemas abaixo, que correspondem a seguinte definição :

  $\{Lemma\ meta\_subst\_var\_neq\ :\ forall\ (x\ y\ :\ tmvar)\ (e\ tm),\ x <> y -> (var\ x)\ \hat{}\{y\ :=\ e\}\ =\ (var\ x).\}$

  $\{Lemma\ meta\_subst\_var\_same\ :\ forall\ (x\ y\ :\ tmvar)\ (e\ :\ tm),$

  $y = x -> (var\ y)\ \hat{}\{x\ :=\ e\}\ =\ e.\}$

  Notar que fizemos uso do lema anteriormente provado, o *meta_subst_not_fv*.

  ```
  intros H H'. rewrite meta_subst_var_same; trivial.
  ```

  ```
  rewrite meta_subst_var_neq; trivial.
  ```

  ```
  rewrite meta_subst_var_same; trivial.
  ```

  ```
  case (atom_eqdec _ z y).  intros H H'.
  ```

  ```
  rewrite meta_subst_var_neq; trivial.
  ```

  ```
  rewrite meta_subst_var_same; trivial.
  ```

  ```
  rewrite meta_subst_not_fv; trivial.
  ```

```
intros H H'. rewrite meta_subst_var_neq; trivial.
repeat rewrite meta_subst_var_neq; trivial.
```

- (* *app* *) No caso da aplicação, não foi necessário utilizar nenhum lema adicional, apenas utilizamos as definições já construídas, e aplica-se as hipóteses de indução.

```
intros s IHs t0 IHt0.
autorewrite with lex.
rewrite IHs; trivial.
rewrite IHt0; trivial.
```

- (* *lam* *) No caso da abstração, introduzimos o *exists*, a fim de eliminar o construtor existencial, e em seguida precisa-se provar que a existência de um z que não faz parte desse conjunto, o que nos leva a noção de conjunto suporte finito, visto na parte teórica da especificação.

```
exists (union (add x (fvar u)) (add y (fvar v))).
intros z z_fresh s IHs.
```

Aplica-se a tática *destruct_neg_union*, definida dentro do módulo sobre propriedades de conjuntos, a fim de dividir a hipótese em duas, ou seja :

~ $In\ z\ (add\ x\ (fvar\ u))$ e ~ $In\ z\ (add\ y\ (fvar\ v))$, e em seguida *destruct_neg_add*, para transformar em $z <> x$ e $z <> y$, e mais ~ $In\ z\ (fvar\ u)$ e ~ $In\ z\ (fvar\ v)$

```
destruct_neg_union z_fresh z_fresh1 z_fresh2.
destruct_neg_add z_fresh1 zx zN.
destruct_neg_add z_fresh2 zy zL.
autorewrite with lex.
rewrite IHs; trivial.
```

Aplica-se a definição da meta-substituição, no caso da abstração, ou melhor :

$\{Theorem\ meta\_subst\_lam\ :\ forall\ x\ y\ t,\ x\ <>\ y\ ->\ \ In\ x\ (fvar\ s) ->$
$meta\_subst\ (lam\ x\ t)\ =\ lam\ x\ (meta\_subst\ t).\}$

```
rewrite meta_subst_lam; trivial.
intro J.
```

Neste caso, aplica-se uma definição considerando as variáveis livres da meta-substituição :

$\{Lemma\ fvar\_meta\_subst\_1\ :\ forall\ (x\ :\ tmvar)\ (M\ N\ :\ tm),$
$Subset\ (fvar\ (M\ \hat{}\{x\ :=\ N\}))\ (union\ (remove\ x\ (fvar\ M))\ (fvar\ N)).\}$

```
assert (J' := fvar_meta_subst_1 _ _ _ _ J).
destruct_union J'.
destruct_remove J' J1 J2; contradiction.
contradiction.
```

- (* *subst* *) Na substituição explícita, procede-se de forma similar ao caso da abstração (lam).

```
exists (union (add x (fvar u)) (add y (fvar v))).
intros z z_fresh s IHs t0 IHt0.
destruct_neg_union z_fresh z_fresh1 z_fresh2.
destruct_neg_add z_fresh1 zx zN.
destruct_neg_add z_fresh2 zy zL.
autorewrite with lex.
rewrite IHs; trivial.
rewrite IHt0; trivial.
rewrite meta_subst_subst; trivial.
intro J.
assert (J' := fvar_meta_subst_1 _ _ _ _ J).
destruct_union J'.
destruct_remove J' J1 J2; contradiction.
contradiction.
Qed.
```

A seguir, apresentamos o $Lema\ 3.5$ (Propriedades básicas), que nos mostra as reduções por substituição, levando-se em consideração os sistemas ex e λ_{ex}. Ele foi dividido em quatro Lemas, conforme a seguir :

$Lemma\ basic\_prop\_2\_ex\ :\ forall\ x\ s\ t\ u,$
$(s \to_{ex} t) \to ((u\ \hat{}\{x\ :=\ s\}) \to^*_{ex} (u\ \hat{}\{x\ :=\ t\})).$

A prova se dá por indução em u, considerando ex, desenvolvendo-a com a mesma idéia do $meta\_subst\_composition$.

$Lemma\ basic\_prop\_2\_lex\ :\ forall\ x\ s\ t\ u,$
$(s \to_{lex} t) \to ((u\ \hat{}\{x\ :=\ s\}) \to^*_{lex} (u\ \hat{}\{x\ :=\ t\})).$

A prova se dá por indução em u, considerando λ_{ex}, desenvolvendo-a com a mesma idéia do $meta\_subst\_composition$.

$Lemma\ basic\_prop\_3\_sys\_ex\ :\ forall\ x\ s\ t\ u,$
$(s \to_{ex} t) \to ((s\ \hat{}\{x\ :=\ u\}) \to_{ex} (t\ \hat{}\{x\ :=\ u\})).$

Neste caso, foi aplicada a indução pré-construída de *Coq* sobre a redução $s \to_{ex} t$, tendo que fazer a prova para todos os subcasos abertos, ou seja, provar para as regras $Var, Gc, App, Lamb\ e\ Comp$.

$Lemma\ basic\_prop\_3\_sys\_Bex\ :\ forall\ x\ s\ t\ u,$
$(s \rightarrow_{Bex} t) \rightarrow ((s\ \hat{}\{x\ :=\ u\}) \rightarrow_{Bex} (t\ \hat{}\{x\ :=\ u\})).$

Neste caso, procede-se como no Lema anterior, no entanto, inclui-se mais a prova da regra B.

Não finalizamos a parte final do Lema, ou seja, $t\{x/u\} \in SN_R$ implica $t \in SN_R$, ficando para trabalhos futuros.

Em seguida, o $Lema\ 3.6$ (Composição Completa para Metatermos), onde a prova se dá por indução em t, da mesma forma que as anteriores, fazendo uso também do $Lema\ 3.3$ para sua conclusão. Mostrando que a substituição explícita implementa a implícita.

$Lemma\ full\_comp\ :\ forall\ x\ t\ u,\ t\ \hat{}[x \mid u]\ \rightarrow^{+}_{ex}\ (t\ \hat{}\{x := u\}).$

Com os Lemas acima provados, daremos inicío a uma abordagem bastante interessante para se chegar à prova completa da confluência : a definição da propriedade Z, que realiza um mapeamento de termos a termos através da utilização da função superdesenvolvimento.

A propriedade Z está definida em *Coq* da seguinte maneira :

$Definition\ Z\_property\ (R\ :\ tm \rightarrow tm \rightarrow Prop)\ :=$
$\{f\ :\ tm \rightarrow tm \mid forall\ t\ u\ :\ tm,\ t \rightarrow_{lex} u \rightarrow (u \rightarrow^{*}_{lex}\ f(t)\ \wedge f(t)\ \rightarrow^{*}_{lex} f(u))\}.$

A função superdesenvolvimento está feita em *Coq* de forma recursiva, definida a seguir :

$Definition\ superdevf\_aux\ (t'\ u'\ :\ tm)\ :\ tm \rightarrow tm\ :=$
$tm\_rec\ tm$
$(app\ t'\ u')$
$(fun\ x\ \Rightarrow\ app\ t'\ u')$
$(fun\ w\ w'\ z\ z'\ \Rightarrow\ app\ t'\ u')$
$(fun\ x\ v\ v'\ \Rightarrow\ meta\_subst\ x\ u'\ v)$
$(fun\ x\ v\ v'\ w\ w'\ \Rightarrow\ app\ t'\ u')$
$(union\ (fvar\ t')\ (fvar\ u')).$

$Definition\ superdevf\ :\ tm \rightarrow tm\ :=$
$tm\_rec\ tm$
dot
$(fun\ x\ \Rightarrow\ (var\ x))$
$(fun\ t\ t'\ u\ u'\ \Rightarrow\ (superdevf\_aux\ t'\ u')\ t)$
$(fun\ x\ t\ t'\ \Rightarrow\ lam\ x\ t')$
$(fun\ x\ t\ t'\ u\ u'\ \Rightarrow\ meta\_subst\ x\ u'\ t')$
$(empty).$

Foi elaborada uma função superdesenvolvimento auxiliar ($superdevf\_aux$) a fim de servir de suporte a implementação da superdesenvolvimento, pois como pode-se observar, no caso da aplicação, efetua-se a chamada da função auxiliar sobre o primeiro termo.

A idéia do funcionamento da função é a seguinte : no caso de dot, retorna o próprio argumento, quando é uma variável retorna a própria variável, no caso da aplicação, considerado o ponto crítico, é interessante observar que sobre o primeiro termo é chamada a função auxiliar, pois, quando esse termo não é uma abstração, ele retorna a própria aplicação, conforme verificado nos construtores da função auxiliar, e quando é uma abstração, retorna a meta-substituição. E continuando na sequência dos construtores, no caso da abstração, propaga, e na substituição explícita, transforma-se na meta-substituição, propagando tanto no primeiro termo, a função, como no segundo, o argumento.

A fim de se chegar à definição do primeiro Lema a utilizar a função superdesenvolvimento, $Lema\ 3.9$, precisamos provar antes, várias propriedades inerentes à especificação do Cálculo λ_{ex} com relação a função superdesenvolvimento, como : conjunto suporte e verificação das suas propriedades, tanto da função auxiliar, como da principal, a fim de constatar que a função está corretamente definida. Sendo assim, partiremos com as definições sobre suporte, considerado uma definição crucial nessa abordagem [36]. A noção de suporte já fora passada na seção sobre lógica nominal, no entanto, é interessante observar sua implementação em *Coq*, uma vez que não conseguimos avançar nas provas sem sua definição. Segue abaixo os casos formalizados :

(∗) $Lemma\ superdevf\_aux\_supp\_dot\ :\ forall\ t\ u,$
$supports\ tmP\ (union\ (fvar\ t)\ (fvar\ u))\ (app\ t\ u).$

(∗) $Lemma\ superdevf\_aux\_supp\_var\ :\ forall\ t\ u,$
$supports\ (tmvarP\ \hat{}\text{->}\ tmP)\ (union\ (fvar\ t)\ (fvar\ u))$
$(fun\ x\ :\ tmvar\ =>\ app\ t\ u).$

(∗) $Lemma\ superdevf\_aux\_supp\_app\ :\ forall\ t\ u,$
$supports\ (tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$(union\ (fvar\ t)\ (fvar\ u))\ (fun\ \_\ \_\ \_\ \_\ :\ tm\ =>\ app\ t\ u).$

(∗) $Lemma\ superdevf\_aux\_supp\_lam\ :\ forall\ t\ u,$
$supports\ (tmvarP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$(union\ (fvar\ t)\ (fvar\ u))$
$(fun\ (x\ :\ tmvar)\ (v\ v'\ :\ tm)\ =>\ meta\_subst\ x\ u\ v).$

(*) $Lemma\ superdevf\_aux\_supp\_subst\ :\ forall\ t\ u,$
$supports\ (tmvarP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$(union\ (fvar\ t)\ (fvar\ u))$
$(fun\ (x\ :\ tmvar)\ (\_\ t'\ \_\ u'\ :\ tm)\ =>\ app\ t\ u).$

De forma similar, é feito para função superdesenvolvimento :

(*) $Lemma\ superdevf\_supp\_dot\ :\ supports\ tmP\ (empty)\ dot.$

(*) $Lemma\ superdevf\_supp\_var\ :\ supports\ (tmvarP\ \hat{}\text{->}\ tmP)\ empty$
$(fun\ x\ :\ tmvar\ =>\ var\ x).$

(*) $Lemma\ superdevf\_supp\_app\ :$
$supports\ (tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$empty\ (fun\ t\ t'\ u\ u'\ :\ tm\ =>\ (superdevf\_aux\ t'\ u'\ t)).$

(*) $Lemma\ superdevf\_supp\_lam\ :$
$supports\ (tmvarP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$empty\ (fun\ (x\ :\ tmvar)\ (t\ t'\ :\ tm)\ =>\ lam\ x\ t').$

(*) $Lemma\ superdevf\_supp\_subst\ :$
$supports\ (tmvarP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP\ \hat{}\text{->}\ tmP)$
$empty\ (fun\ (x\ :\ tmvar)\ (\_\ t'\ \_\ u'\ :\ tm)\ =>\ meta\_subst\ x\ u'\ t').$

Observar que os suportes estão definidos em conformidade com a definição dos construtores da função auxiliar e principal, ficando razoavelmente fácil sua compreensão.

Após a definição dos suportes, partimos para verificar se as funções foram corretamente definidas, provando as seguintes propriedades :

(*) $Lemma\ superdevf\_aux\_dot\ :\ forall\ t\ u,$
$superdevf\_aux\ t\ u\ dot\ =\ app\ t\ u.$
A aplicação da função em dot, retorna a própria aplicação.

(*) $Lemma\ superdevf\_aux\_var\ :\ forall\ x\ t\ u,$
$superdevf\_aux\ t\ u\ (var\ x)\ =\ app\ t\ u.$
A aplicação da função em var, retorna a própria aplicação.

(∗) $Lemma\ superdevf\_aux\_app\ :\ forall\ t\ u\ w\ z,$
$superdevf\_aux\ t\ u\ (app\ w\ z)\ =\ app\ t\ u.$
A aplicação da função em app, retorna a própria aplicação.

(∗) $Lemma\ superdevf\_aux\_lam\ :\ forall\ t\ u\ x\ v,$
$(\tilde{}\ In\ x\ (union\ (fvar\ t)\ (fvar\ u)))->$
$superdevf\_aux\ t\ u\ (lam\ x\ v)\ =\ meta\_subst\ x\ u\ v.$
A aplicação da função em lam, retorna a meta-substituição.

(∗) $Lemma\ superdevf\_aux\_subst\ :\ forall\ t\ u\ x\ v\ w,$
$(\tilde{}\ In\ x\ (union\ (fvar\ t)\ (fvar\ u)))->$
$superdevf\_aux\ t\ u\ (subst\ x\ w\ v)\ =\ app\ t\ u.$
A aplicação da função em $subst$, retorna a própria aplicação.

De forma similar, é feito para função superdesenvolvimento :

(∗) $Theorem\ superdevf\_dot\ :\ superdevf\ dot\ =\ dot.$
A aplicação da função em dot, retorna o próprio dot.

(∗) $Theorem\ superdevf\_var\ :\ forall\ x,\ superdevf\ (var\ x)\ =\ (var\ x).$
A aplicação da função em var, retorna o próprio var.

(∗) $Theorem\ superdevf\_app\_dot\ :\ forall\ u,$
$superdevf\ (app\ dot\ u)\ =\ app\ dot\ (superdevf\ u).$
No caso da aplicação, quando o primeiro termo é dot, retorna dot aplicado ao segundo termo.

(∗) $Theorem\ superdevf\_app\_var\ :\ forall\ x\ u,$
$superdevf\ (app\ (var\ x)\ u)\ =\ app\ (var\ x)\ (superdevf\ u).$
No caso da aplicação, quando o primeiro termo é var, retorna var aplicado ao segundo termo.

(∗) $Theorem\ superdevf\_app\_app\ :\ forall\ s\ t\ u,$
$superdevf\ (app\ (app\ s\ t)\ u)\ =\ app\ (superdevf\ (app\ s\ t))\ (superdevf\ u).$
No caso da aplicação, quando o primeiro termo é uma aplicação app, retorna a própria aplicação da função e aplicada ao segundo termo.

(∗) $Lemma\ superdevf\_app\_subst\ :\ forall\ x\ s\ t\ u,$

$superdevf\ (app\ (subst\ x\ s\ t)\ u)\ =\ app\ (superdevf\ (subst\ x\ s\ t))\ (superdevf\ u).$

No caso da aplicação, quando o primeiro termo é uma substituição explícita $subst$, retorna a própria aplicação da função e aplicada ao segundo termo.

(∗) $Theorem\ superdevf\_app\_lam\ :\ forall\ x\ v\ u,$

$superdevf\ (app\ (lam\ x\ v)\ u)\ =\ meta\_subst\ x\ (superdevf\ u)\ v.$

No caso da aplicação, quando o primeiro termo é uma abstração lam, retorna a meta-substituição, com a função aplicada ao segundo termo, o argumento da meta.

(∗) $Theorem\ superdevf\_lam\ :\ forall\ x\ t,$

$superdevf\ (lam\ x\ t)\ =\ lam\ x\ (superdevf\ t).$

No caso da função aplicada a uma abstração, apenas propaga sobre o termo.

(∗) $Theorem\ superdevf\_subst\ :\ forall\ x\ t\ u,$

$superdevf\ (subst\ x\ t\ u)\ =\ meta\_subst\ x\ (superdevf\ u)\ (superdevf\ t).$

No caso da função aplicada à substituição explícita, retorna a meta-substituição, sendo a função aplicada tanto no primeiro termo (função), como no segundo (argumento).

Este trabalho está em constante desenvolvimento e o código fonte com a especificação e provas encontra-se disponível em :

$http://www.cic.unb.br/\tilde{}flavio/msc/lex\_conf.tar.gz$

Capítulo 7

Conclusão

A área de verificação formal tem crescido rapidamente nos últimos anos devido a um grande interesse e importância de se obter sistemas livres de erros. Isto é particularmente importante em sistemas de natureza crítica, como por exemplo, no controle de tráfego aéreo, monitoramento em UTI de hospitais, sistemas financeiros, etc.

Com base na garantia de um sistema totalmente correto, propomos realizar a especificação do cálculo λ_{ex} em *Coq*, onde esta formalização mecânica (especificação e provas de propriedades) nos permite uma análise minuciosa do objeto em estudo, onde todos os detalhes precisam ser considerados. Tal nível de detalhamento permite a detecção e correcção de erros de forma consistente. Dada a crescente complexidade dos sistemas computacionais modernos, torna-se urgente o provimento de garantias de que tais sistemas são corretos e seguros. Tais garantias podem ser fornecidas via formalizacão mecânica utilizando lógica de ordem superior, que em nosso caso, utilizou como ferramenta de formalização o assistente de provas *Coq*.

Antes de partirmos para os estudos da verificação mecânica do cálculo citado, realizamos um estudo teórico sobre o λ-Cálculo, que além de ser um sistema de reescrita de ordem superior, é o primeiro sistema de reescrita desenvolvido do ponto de vista computacional, considerado o fundamento teórico do paradigma de programação funcional. Em seguida fornecemos uma visão sobre cálculos de substituições explícitas, uma área em que as pesquisas têm evoluído bastante nos últimos anos devido a um grande interesse em se poder utilizar formalismos que sejam fiéis ao λ-Cálculo, ou seja, preservando todas as propriedades do sistema original.

Introduzimos também, como suporte à nossa implementação, os principais conceitos e fundamentos da lógica nominal, que nos permite a especificação da α-conversão de uma forma natural, e bem próxima à que os teóricos utilizam.

Paralelamente aos estudos da especificação, onde foi feita utilizando-se de uma adaptação da gramática de *Nominal Reasoning in Coq* [3], desenvolvemos a formalização em *Coq* de lemas e propriedades importantes relativas a estratégia de prova adota em [27]. De fato, esta prova se baseia em mostrar que o λ_{ex} satisfaz a propriedade Z [34], o que nos permitiu provar diversos resultados sobre o suporte da função superdesenvolvimento que exerce um papel central na prova da confluência do λ_{ex}

A partir desse trabalho, podemos estender seus resultados a outros tipos de cálculos. Além disso, uma formalização desse tipo pode ser útil na construção de sistemas baseados em cálculos de substituições explícitas, e em particular no λ_{ex}, onde uma possível versão em notação de *de Bruijn* é de particular interesse em implementações reais.

Referências

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit Substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. 15, 16

[2] P. Aczel. A general church-rosser theorem, 1978. Unpublished note, University of Manchester. 25

[3] B. Aydemir, A. Bohannon, and S. Weirich. Nominal reasoning techniques in coq. *ENTCS*, pages 1–9, 2006. v, vi, 2, 34, 36, 52

[4] H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984. 3, 4, 13

[5] H. P. Barendregt. λ-calculi with types. *Handbook of Logic in Computer Science*, II, 1992. 7

[6] H. P. Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Simbolic Logic*, 3(3):181–215, 1997. 3

[7] H. P. Barendregt and H. Geuvers. Proof-assistants using dependent type systems. In *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier and MIT Press, 2001. 26, 27

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. 26, 30, 31, 32

[9] R. Bloo and K. Rose. Preservation of strong normalization in named lambda calculi with explicit substitution and garbage collection. pages 62–72. In Computer Science in the Netherlands (CSN), 1995. 15

[10] J. Campos. Guest Editorial Special Section on Formal Methods in Manufacturing. In *IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS*, volume 6, Zaragoza, Spain, 2010. IEEE Computer Society. 1

[11] V. Carreño and C. Muñoz. Safety verification of the Small Aircraft Transportation System concept of operations. In *Proceedings of the AIAA 5th Aviation, Technology, Integration, and Operations Conference, AIAA-2005-7423*, Arlington, Virginia, 2005. 1

[12] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, 1941. 3

[13] A. Church and J. B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39:472–482, 1936. 3

[14] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.1*. INRIA-Rocquencourt, 2007. 26, 27

[15] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958. 13

[16] H.B Curry. Functionality in combinatory logic. *Proc.Nat.Acad.Science USA 20*, pages 584–590, 1934. 13

[17] N.G. de Bruijn. Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indag. Mat.*, 34(5):381–392, 1972. 2

[18] H. M. Friedman. Adventures in the verification of mathematics. In *Computer Science Colloquium*. Ohio State University, 2006. 26

[19] T. Hardin and J.-J. Lévy. A Confluent Calculus of Substitutions. *France-Japan Artificial Intelligence and Computer Science Symposium*, December 1989. 16

[20] J. R. Hindley. *Basic simple type theory*. Cambridge University Press, 1997. 13

[21] J. R. Hindley and J. P. Seldin. *lambda-calculus and combinators, an Introduction* . Cambridge University Press, 2008. 13

[22] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq Proof Assistant A Tutorial – Version V8.1. Technical report, INRIA, February 2007. 29

[23] F. Kamareddine and A. Ríos. A lambda-calculus ‘a la de bruijn with explicit substitutions. In *PLILP*, pages 45–62, 1995. 15

[24] F. Kamareddine and A. Ríos. Extending a lambda-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *J. Funct. Program.*, 7(4):395–420, 1997. 15

[25] D. Kesner. The theory of calculi with explicit substitutions revisited. In *Proc. 16th Annual Conference on Computer Science and Logic (CSL-07)*, LNCS, Lausanne, Switzerland, 2007. Springer-Verlag. 14, 15, 16

[26] D. Kesner. Perpetuality for full and safe composition (in a constructive setting). In *To appear in Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP 2008), Track B*, Reykjavik, Iceland,, 2008. 2

[27] D. Kesner. A theory of explicit substitutions with safe and full composition. *Logical Methods in Computer Science*, 5(3):1–29, 2009. 2, 15, 16, 18, 19, 22, 23, 25, 52, 58, 59, 60

[28] S. Kleene and B. Rosser. The inconsistency of certain formal logics. *Annals of Math.*, 36(2):630–636, 1935. 3

[29] S. C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936. 3

[30] P. Letouzey. Coq Extraction, an Overview. In C. Dimitracopoulos A. Beckmann and B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008. 32

[31] R. Lins. A new formula for the execution of categorical combinators. In *In 8th Conference on Automated Deduction (CADE)*, volume 230, pages 89–98. Lecture Notes in Computer Science, 1986. 15

[32] R. Lins. Partial categorical multi-combinators and Church Rosser theorems. In *Technical Report 7/92, Computing Laboratory, University of Kent at Canterbury*, 1992. 15

[33] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate in Proceedings of TLCA'95. *LNCS*, 902, 1995. 16

[34] V. V. Oostrom. Z, 2007. Available at `http://www.phil.uu.nl/ oostrom/publication/rewriting.html for slides`. 2, 25, 52, 61

[35] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(1):165–193, 2003. 33

[36] A. M. Pitts. Alpha-structural recursion and induction (extended abstract). In J.Hurd and T.Melham, editors, *Theorem Proving in Higher Order Logics - TPHOLs 2005*, volume 3603 of *LNCS*, pages 17–34. Springer, 2005. 36, 48

[37] A. M. Pitts. Alpha-Structural Recursion and Induction. *Journal of the ACM*, 53(3):459–506, 2006. 39

[38] F. V. Raamsdonk. Confluence and superdevelopments. In In Claude Kirchmer, editor, *5th International Conference on Rewriting Techniques and Applications(RTA)*, volume 690 of *LNCS*, pages 168–182. Springer-Verlag, 1993. 25

[39] K. Rose. Explicit cyclic substitutions. In *In MichaÂĺel Rusinowitch and Jean-Luc RÂťemy, editors, Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems (CTRS)*, volume 656, pages 36–50. Lecture Notes in Computer Science, 1992. 15

[40] G. Swamy. Formal Verification of Digital Systems. In *Tenth International Conference on VLSI Design: VLSI in Multimedia Applications*, Boston, MA, USA, 2007. IEEE Computer Society. 1

[41] A. M. Turing. A set of postulates for the foundation of logic (second paper). *J.Symbolic Logic 2*, pages 153–163, 1937. 3

[42] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, Cambridge, revised edition, 1925–1927. Three volumes. The first edition was published 1910–1913. 3

Apêndice A

Apêndice A. Prova da Confluência

Lema A.1. *[27] Sejam t, u metatermos. Então $t^\circ u^\circ \to^*_{\lambda_{ex}} (tu)^\circ$.*

Prova. Se t° não é uma abstração, então $t^\circ u^\circ = (tu)^\circ$. Se $t^\circ = \lambda_y.s$, então $t^\circ u^\circ = (\lambda_y.s)u^\circ \to_B s[y/u^\circ] \to^*_{ex} (Lema\ 3.6)\ s\{y/u^\circ\} = (tu)^\circ$. □

Lema A.2. *[27] Sejam t, u metatermos. Então $t^\circ\{x/u^\circ\} \to^*_{\lambda_{ex}} t\{x/u\}^\circ$.*

Prova. A prova é por indução em t. Supor $t = vw$.

- Se v° não é uma abstração, então

  $(vw)^\circ\{x/u^\circ\} = v^\circ\{x/u^\circ\}w^\circ\{x/u^\circ\} \overset{IH}{\to}^*_{\lambda_{ex}} v\{x/u\}^\circ w\{x/u\}^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)$ $(vw)\{x/u\}^\circ$

- Se $v^\circ = \lambda_z.r$, então a IH nos dá $v^\circ\{x/u^\circ\} = (\lambda_z.r)\{x/u^\circ\} \to^*_{\lambda_{ex}} v\{x/u\}^\circ$ tal que $v\{x/u\}^\circ = \lambda_z.s$ onde $r\{x/u^\circ\} \to^*_{\lambda_{ex}} s$. Como consequência,

  $(vw)^\circ\{x/u^\circ\} = r\{z/w^\circ\}\{x/u^\circ\} =_e (Lema\ 3.4)\ r\{x/u^\circ\}\{z/w^\circ\{x/u^\circ\}\} \to^*_{\lambda_{ex}}$
  $s\{z/w^\circ\{x/u^\circ\}\} \overset{IH}{\to}^*_{\lambda_{ex}} \&\ Lema\ 3.5)\ s\{z/w\{x/u\}^\circ\} = (v\{x/u\}w\{x/u\})^\circ = (vw)\{x/u\}^\circ$

  O caso $t = v[y/w]$ também usa a hipótese de indução e o $Lema\ 3.4$. Todos os outros casos são razoavelmente simples. □

Lema A.3. *[27] Seja t um metatermo. Então $t \to^*_{\lambda_{ex}} t^\circ$.*

Prova. A prova é por indução em t. Os casos interessantes são os seguintes.

- $t = uv$: Então $uv \overset{IH}{\to}^*_{\lambda_{ex}} u^\circ v^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)\ (uv)^\circ = t^\circ$.
- $t = u[x/v]$: Então $u[x/v] \overset{IH}{\to}^*_{\lambda_{ex}} u^\circ[x/v^\circ] \to^*_{ex} (Lema\ 3.6)\ u^\circ\{x/v^\circ\}$ $\to^*_{\lambda_{ex}} (Lema\ A.2)\ u\{x/v\}^\circ$.

  Todos os outros casos são razoavelmente simples. □

Lema A.4. *[27] Sejam t, u metatermos. Se $t \to_{Bx} u$, então $u \to^*_{\lambda_{ex}} t^\circ \to^*_{\lambda_{ex}} u^\circ$.*

Obs : A relação de redução $\to_{\lambda_{ex}}$ é gerada pela relação de reescrita $\to_{Bx}$, em que $\to_{Bx}$ possui todas as regras de reescrita do $\to_{\lambda_{ex}}$, exceto a equação C.

Prova. Por indução em $t \to_{Bx} u$.

- Se $t = \lambda_x.r \to_{Bx} \lambda_x.s = u$, onde $r \to_{Bx} s$, então a propriedade se mantém pela IH.
- Se $t = r[x/v] \to_{Bx} s[x/v] = u$, onde $r \to_{Bx} s$, então

  $u = s[x/v] \overset{IH}{\to}^*_{\lambda_{ex}} r^\circ[x/v] \to^*_{\lambda_{ex}} (Lema\ A.33)\ r^\circ[x/v^\circ] \to^*_{ex} (Lema\ 3.6)$
  $r^\circ\{x/v^\circ\} = t^\circ \overset{IH}{\to}^*_{\lambda_{ex}} \quad s^\circ\{x/v^\circ\} = s[x/v]^\circ = u^\circ$
- Se $t = v[x/r] \to_{Bx} v[x/s] = u$, onde $r \to_{Bx} s$, então proceder como no caso anterior.
- Se $t = rv \to_{Bx} sv = u$, onde $r \to_{Bx} s$, então $sv \overset{IH}{\to}^*_{\lambda_{ex}} \ r^\circ v \to^*_{\lambda_{ex}} (Lema\ A.3)$
  $r^\circ v^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)\ (rv)^\circ$.

  Para segunda parte da afirmação existem dois casos :

  - Se r° não é uma abstração, então $(rv)^\circ = r^\circ v^\circ \overset{IH}{\to}^*_{\lambda_{ex}}$
  $s^\circ v^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)\ (sv)^\circ$.

  - Se $r^\circ = \lambda_z.w$, então a IH $r^\circ \to^*_{\lambda_{ex}} s^\circ$ implica $s^\circ = \lambda_z.q$, onde $w \to^*_{\lambda_{ex}} q$.
  Nós concluímos que $(rv)^\circ = w\{z/v^\circ\} \to^*_{\lambda_{ex}} (Lema\ 3.5)\ q\{z/v^\circ\} = (sv)^\circ$.
- Se $t = vr \to_{Bx} vs = u$, onde $r \to_{Bx} s$, então $vs \overset{IH}{\to}^*_{\lambda_{ex}}$
  $vr^\circ \to^*_{\lambda_{ex}} (Lema\ A.3)\ v^\circ r^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)\ (vr)^\circ$.

  Para segunda parte da afirmação, existem dois casos :

  - Se v° não é uma abstração, então $(vr)^\circ = v^\circ r^\circ \overset{IH}{\to}^*_{\lambda_{ex}} \ v^\circ s^\circ = (vs)^\circ$.

  - Se $v^\circ = \lambda_y.w$, então $(vr)^\circ = w\{y/r^\circ\} \overset{IH}{\to}^*_{\lambda_{ex}} \ \&\ Lema\ 3.5)\ w\{y/s^\circ\} = (vs)^\circ$.
- Se $t = x[x/v] \to_{Var} v = u$, então $x[x/v]^\circ = x\{x/v^\circ\} = v^\circ$. Nós concluímos, uma vez que $v \to^*_{\lambda_{ex}} v^\circ$ mantém-se pelo $Lema\ A.3$.
- Se $t = r[x/v] \to_{Gc} r = u$, então $r[x/v]^\circ = r^\circ\{x/v^\circ\} = (Lema\ 3.3)\ r^\circ$.
  Nós concluímos, uma vez que $r \to^*_{\lambda_{ex}} r^\circ$ mantém-se pelo $Lema\ A.3$.

- Se $t = (rs)[x/v] \to_{App} r[x/v]s[x/v] = u$, então

  $u \to^*_{\lambda_{ex}} (Lema\ A.3)\ r^\circ[x/v^\circ]s^\circ[x/v^\circ] \to^*_{ex} (Lema\ 3.6)$
  $r^\circ\{x/v^\circ\}s^\circ\{x/v^\circ\} = (r^\circ s^\circ)\{x/v^\circ\} \to^*_{\lambda_{ex}} (Lema\ 3.5\ \&\ Lema\ A.1)$
  $(rs)^\circ\{x/v^\circ\} = (rs)[x/v]^\circ = t^\circ$.

  Para segunda parte existem dois casos :

  - Se r° não é uma abstração, então
  $t^\circ = r^\circ\{x/v^\circ\}s^\circ\{x/v^\circ\} = r[x/v]^\circ s[x/v]^\circ \to^*_{\lambda_{ex}} (Lema\ A.1)\ (r[x/v]s[x/v])^\circ = u^\circ$

  - Se $r^\circ = \lambda_y.q$, então $r[x/v]^\circ = \lambda_y.q\{x/v^\circ\}$, tal que
  $t^\circ = (rs)[x/v]^\circ = (rs)^\circ\{x/v^\circ\} = q\{y/s^\circ\}\{x/v^\circ\} = e\ (Lema\ 3.4)$
  $q\{x/v^\circ\}\{y/s^\circ\{x/v^\circ\}\} = q\{x/v^\circ\}\{y/s[x/v]^\circ\} =$
  $(r[x/v]s[x/v])^\circ = u^\circ$

- Se $t = (\lambda_y.r)[x/v] \to_{Lamb} \lambda_y.r[x/v] = u$, então $(\lambda_y.r)[x/v]^\circ = \lambda_y.r^\circ\{x/v^\circ\}$.
  Nós temos $u = \lambda_y.r[x/v] \to^*_{\lambda_{ex}} (Lema\ A.3)\ \lambda_y.r^\circ[x/v^\circ] \to^*_{ex} (Lema\ 3.6)\ \lambda_y.r^\circ\{x/v^\circ\} = t^\circ = u^\circ$

- Se $t = r[x/v][y/w] \to_{Comp} r[y/w][x/v[y/w]] = u$, então
  $u = r[y/w][x/v[y/w]] \to^*_{\lambda_{ex}} (Lema\ A.3)\ r^\circ[y/w^\circ][x/v^\circ[y/w^\circ]] \to^*_{\lambda_{ex}} (Lema\ 3.6\ \&\ Lema\ 3.5)\ r^\circ\{y/w^\circ\}\{x/v^\circ\{y/w^\circ\}\} =_e (Lema\ 3.4)\ r^\circ\{x/v^\circ\}\{y/w^\circ\} = t^\circ$

  Uma vez que $u^\circ = r^\circ\{y/w^\circ\}\{x/v^\circ\{y/w^\circ\}\}$, então temos $t^\circ \to^*_{\lambda_{ex}} u^\circ$. □

Lema A.5. *[27] Sejam t, u metatermos tal que $t =_e u$. Então,*

- *Se $r =_e s$, então $t\{x/r\} =_e u\{x/s\}$.*
- $t^\circ =_e u^\circ$.

Prova. Supor $t =_e u$ mantém-se em n passos. Ambas as propriedades podem ser simultâneamente provadas por indução no par lexicográfico (n, t). □

Corolário A.6. *(Propriedade-Z). Sejam t, u metatermos. Se $t \to_{\lambda_{ex}} u$, então $u \to^*_{\lambda_{ex}} t^\circ \to^*_{\lambda_{ex}} u^\circ$.*

Prova. Seja $t =_e r \to_{Bx} s =_e u$. Pelo $Lema\ A.4\ r \to^*_{\lambda_{ex}} s^\circ \to^*_{\lambda_{ex}} r^\circ$ e pelo $Lema\ A.5$ $t^\circ =_e r^\circ\ e\ s^\circ =_e u^\circ$. Assim concluímos $t \to^*_{\lambda_{ex}} u^\circ \to^*_{\lambda_{ex}} t^\circ$. □

Corolário A.7. *(Confluência). A relação de redução $\to_{\lambda_{ex}}$ é confluente nos metatermos.*

Teorema A.8. *[34](Z implica Confluência) Se $\rightarrow_R$ satisfaz a propriedade Z, então $\rightarrow_R$ é confluente.*

Prova. Supor que $\_^\circ$ é uma função que satisfaz a propriedade-Z para R.

- Definir $a^\bullet := a$ se a é uma forma normal de R, $a^\bullet := a^\circ$ o contrário.
- Provar que $\_^\bullet$ também satisfaz a propriedade-Z para $\rightarrow_R$.

  Se $a \rightarrow_R b$, então $b \rightarrow_R^* a^\circ \rightarrow_R^*$ b$^\circ$ pela hipótese e $a^\bullet = a^\circ$ pelo item (1) tal que $b \rightarrow_R^* a^\bullet$. Se b é uma forma normal de R, então $b^\bullet = b = a^\circ = a^\bullet$ tal que $a^\bullet \rightarrow_R^* b^\bullet$. Se b não é uma forma normal de R, então $b^\bullet = b^\circ$ tal que também $a^\bullet = a^\circ \rightarrow_R^* b^\circ = b^\bullet$.
- Provar que $a \rightarrow_R^* a^\bullet$.

  Se a é uma forma normal de R, então $a^\bullet = a$, assim como feito. Ao contrário, existe um b tal que $a \rightarrow_R b$, tal que pelo item (2) nos dá $b \rightarrow_R^* a^\bullet$ e assim $a \rightarrow_R^* a^\bullet$.
- Provar que $a \rightarrow_R^* b$ implica $a^\bullet \rightarrow_R^* b^\bullet$.

  Por indução sobre o número n de passos de a para b. Se n=0, então $a = b$ e $a^\bullet = b^\bullet$. Se $n > 0$, então $a \rightarrow_R c \rightarrow_R^* b$, onde $c \rightarrow_R^* b$ mantém-se em n-1 passos. Item (2) e a IH fornece $a^\bullet \rightarrow_R^* c^\bullet \rightarrow_R^* b^\bullet$.
- Concluir confluência de $\rightarrow_R$.

  Seja $t \rightarrow_R^* t1$ e $t \rightarrow_R^* t2$. Nós queremos mostrar que existe um $t3$ tal que $t1 \rightarrow_R^* t3$ e $t2 \rightarrow_R^* t3$. Nós procedemos por indução sobre o número n de passos de t a $t2$. Se $n = 0$, então $t = t2$ e tomamos $t3 = t1$ tal como feito. Se $n > 0$, então $t \rightarrow_R u \rightarrow_R^* t2$, com n-1 passos de u para $t2$. Pelo item (2) $u \rightarrow_R^* t^\bullet$ e pelo item (4) $t^\bullet \rightarrow_R^* t1^\bullet$ tal que $u \rightarrow_R^* t1^\bullet$. Pelo item (3) $t1 \rightarrow_R^* t1^\bullet$. Agora, $u \rightarrow_R^* t1^\bullet$ e $u \rightarrow_R^* t2$ mantém-se em n-1 passos, então fechamos o diagrama pela IH. □