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

Verificação de Propriedades do Cálculo λex em Coq

Washington Luís Ribeiro de Carvalho Segundo

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. Mario Roberto Folhadela Benevides — COPPE/UFRJ
Prof. Dr. Mauricio Ayala Rincón — CIC/UnB

CIP — Catalogação Internacional na Publicação

Segundo, Washington Luís Ribeiro de Carvalho.

Verificação de Propriedades do Cálculo λex em Coq / Washington Luís Ribeiro de Carvalho Segundo. Brasília : UnB, 2010.
71 p. : il. ; 29,5 cm.

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

1. verificação formal, 2. cálculos de substituições explícitas, 3. cálculo λex

CDU 004.4

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

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

Verificação de Propriedades do Cálculo λex em Coq

Washington Luís Ribeiro de Carvalho Segundo

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

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

Prof. Dr. Mario Roberto Folhadela Benevides
COPPE/UFRJ

Prof. Dr. Mauricio Ayala Rincón
CIC/UnB

Prof. Dr. Mauricio Ayala Rincón
Coordenador do Mestrado em Informática

Brasília, 13 de julho de 2010

Dedico esse trabalho em memória de meu avô, *José Gonçales*.

Agradecimentos

É algo bastante comum, que um aluno agradeça em primeiro lugar seu mestre. De fato, não há como não reconhecer, de "primeira mão", a fantástica dedicação, paciência e companheirismo empenhado pelo meu orientador, o professor Dr. Flávio de Moura. Dedico também meus agradecimentos as contribuições e sugestões efetuadas pelos professores Dr. Maurício Ayala e Dr. Mario Benevides, na realização da versão final desse documento. Ademais, devo agradecer meus amigos colegas, em especial Andréia Avelar e Flávio Barros, pela parceria na luta de cada dia. A minha rosa, linda Iraciara, agradeço a paciência prestada, todas a vezes que meu humor não ia bem, após um dia cheio. Aos meus tios daqui, Manoel e Cristina, não sei como expressar a gratidão pelo suporte que me foi dado, desde os primeiros dias de residência nessa capital. Ao meu tio Ricardo e minha avó Derly, agradeço o amor incondicional e a força dada a todo momento. Aos meus pais, Washington e Denise, agradeço minha existência, minha educação e tudo que pode haver de bom em mim. A Deus, a conciência regente de todas as coisas, agradeço o fascinante e grande universo, com sua riqueza de detalhes e mistérios, que nos foi legado.

Resumo

O cálculo λex representa uma solução importante dentro da classe de *cálculos de substituições explícitas* que lidam com *"nomes"*, em oposição aqueles que codificam suas variáveis por índices. Delia Kesner obteve, através de um conjunto de provas construtivas, demostrações das importantes propriedades do λex. Dentre elas, destacamos a PSN, isso é, a Preservação da Normalização Forte, cuja demonstração faz uso de uma estratégia de redução perpétua, que permitiu uma caracterização indutiva do conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$. Estendemos a especificação em Coq, já realizada para o cálculo λ, de B. Aydemir *et al*, e que utiliza lógica nominal para construção de princípios de indução e recursão α-estrutural. Dessa forma nossa especificação inclui a substituição explícita ($s[x/t]$) na gramática de termos. Avançamos definindo os sistemas de reescrita e as relações de redução do λex, e concluímos por formalizar alguns resultados para o cálculo, a saber: a FC (Composição Completa), a SIM (Simulação de um passo da β-redução) e ainda outros que caminham para a formalização da PSN.

Palavras-chave: verificação formal, cálculos de substituições explícitas, cálculo λex

Abstract

The λex-calculus represents an important solution among all the class of *explicit substitutions calculi* that deal with *"names"*, as opposed to those that encode variables by indices. Delia Kesner developed the proofs, through a set of constructive ones, of important properties of the λex calculus. Among them, we highlight the PSN property, that is, the Preservation of Strong Normalization, whose proof uses a perpetual reduction strategy which allowed an inductive characterization of the set $\mathcal{SN}_{\lambda\mathtt{ex}}$. We extended the specification already done in Coq for the λ-calculus by B. Aydemir *et al*, using nominal logic to build principles of α-structural induction and recursion. In this way our specification includes the explicit substitution ($s[x/t]$) in the grammar of the terms. We go foward by defining the rewriting systems and the reduction relations for the λex and we conclude by formalizing some results for this calculus, as follows: The FC (Full Composition), SIM (Simulation of One Step of β-Reduction) and others that go in the direction of the formalization of the PSN.

Keywords: formal verification, explicit substitutions calculi, λex-calculus

Sumário

1 Introdução **1**

2 Fundamentos **6**

2.1 O sistema Coq 6

2.2 Lógica formal 10

2.3 Sistemas de reescrita 13

2.4 Lógica nominal 16

3 O Cálculo λ **25**

3.1 Classes de α-equivalência ($\Lambda/_{=_\alpha}$) 27

3.2 Recursão e indução α-estrutural 30

3.3 A operação de meta-substituição definida sobre $\Lambda/_{=_\alpha}$ 32

3.4 O conjunto $\mathcal{SN}_\beta$ 33

4 Cálculos de substituições explícitas **36**

4.1 Cálculos em notação de *de Bruijn* ($\lambda\sigma$, $\lambda\sigma_{\Uparrow}$) 36

4.2 Cálculos com *nomes* (λx, λex, λes) 39

4.3 Uma gramática auxiliar (λex') 45

5 Propriedades do λex **48**

5.1 Composição completa 49

5.2 Simulação de um passo da β-redução 51

5.3 Perpetualidade 52

6 Conclusão e trabalhos futuros **56**

Referências **58**

Lista de Figuras

1.1 *Organização do código fonte.* 5

2.1 *Cálculo λ simplesmente tipado.* 7
2.2 *Demonstração de* `all_perm` *no Coq.* 8
2.3 *Definição indutiva do tipo* nat. 8
2.4 *Tipo indutivo* `tree`*, função recursiva* `n_leaf` *e lema* `n_leaf_gt_zero`. . . . 9
2.5 *Exemplo de árvore binária de números naturais.* 9
2.6 *Exemplo de representação em árvore do termo $f(g(x, y), c)$.* 13
2.7 *Fecho transitivo de uma relação binária.* 14
2.8 *Fecho reflexivo-transitivo de uma relação binária.* 14
2.9 *Forma normal.* 15
2.10 *Átomos.* 18
2.11 *Assinatura nominal do cálculo λ.* 19
2.12 *α-equivalência.* 19
2.13 *Funções recursivas: Conjunto de Átomos e Substituição.* 20
2.14 *Transposições ou Swaps.* 21
2.15 *Ações de permutações.* 21
2.16 *Conjunto Suporte.* 22
2.17 *Predicado Fresh.* 22

3.1 *Definição da regra β.* 27
3.2 *α-equivalência no cálculo λ.* 28
3.3 *Inclusão progressiva dos conjuntos* `Phi n`. 29
3.4 *Conjunto* `Phi`. 29
3.5 *Termos bem formados e construtores do cálculo λ.* 29
3.6 *Teorema de α-equivalência para o construtor abstração.* 30
3.7 *Teorema de indução α-estrutural para o Cálculo λ.* 30

3.8 *Funções indexadas pelos construturas do Cálculo* λ. 31
3.9 *Condição FCB para o construtor abstração.* 31

4.1 *Regras de reescrita do Cálculo* $\lambda\sigma$. 37
4.2 *Conjunto* `Phi` *acrescido do construtor da substituição explícita.* 39
4.3 *Teorema de* α*-equivalência para a abstração e a substituição explícita.* . . . 39
4.4 *Regras de reescrita do Cálculo* λx. 40
4.5 *Equação* `C`. 41
4.6 *Regras de reescrita do cálculo* λex. 41
4.7 *Conjunto de regras do* λex *no Coq.* 42
4.8 *Sistema de reescrita* `ex`. 43
4.9 *Sistema de reescrita* `Bex`. 43
4.10 *Relação de redução* λ`ex`. 43
4.11 *Sistema de regras do cálculo* λ*es.* 44
4.12 *Tabela comparativa.* 45
4.13 *Teorema de indução* α*-Estrutural para o cálculo* λex. 45
4.14 λex *e* λex'. 46
4.15 *Bijeção entre* `Phi` *e* f(`Phi`). 47
4.16 *Meta-substituição no* λex*, via da gramática auxiliar.* 47

5.1 *Propriedades básicas.* 48
5.2 *Compatibiliade da estratégia perpétua com a relação* `lex+`. 53
5.3 *Propriedade* IE. 53
5.4 *Teorema da perpetualidade.* 53
5.5 *Caracterização indutiva do conjunto* $\mathcal{SN}_{\lambda\mathtt{ex}}$. 55
5.6 *Preservação da normalização forte.* 55

Capítulo 1

Introdução

O cálculo λex representa uma solução importante dentro da classe de *cálculos de substituições explícitas* [1] que lidam com *"nomes"*, em oposição aqueles que codificam suas variáveis por índices. Ao mesmo tempo, uma abordagem formal de suas propriedades passa pela definição e demostração de princípios de indução e recursão sobre classes de equivalências, isso é, aquelas definidas pela relação de α-equivalência entre termos. Optamos por realizar esse trabalho no sistema *Coq* [2], e a apresentação de tal solução é dada nesse documento, juntamente com a verificação formal de algumas das propriedades do λex.

Como motivação preliminar, gostaríamos de inserir uma breve citação, bastante pertinente, retirada do livro de Thomas Hobbes[1]; *Leviatã ou Matéria, Forma e Poder de um Estado Eclesiástico e Civil*:

*"(...) Os gregos têm uma só palavra, **logos**, para **linguagem** e **razão**; não que eles pensassem que não havia linguagem sem razão, mas sim que não havia raciocínio sem linguagem."*

Tal afirmação gera uma reflexão sobre o real significado da palavra *"razão"*. De fato, em *Lógica Formal*, tudo tem como base a interpretação de símbolos, isto é, a definição de uma *gramática* e *semânticas* adjacentes. Em conjunto, esses dois conceitos dão origem a uma *linguagem* que viabilizará o estabelecimento de *regras de dedução* ou *procedimentos computacionais*, o que mais tarde irá resultar na construção de *provas matemáticas* e *algoritmos computacionais*. Mas o que é uma prova matemática? Como provas podem ser justificadas? Existem limitações para o que é possível provar? Máquinas podem fornecer provas matemáticas?

Ao final século XIX surgiram diversos avanços na obtenção de respostas para essas questões. Os pensadores que se ocuparam com esses tópicos: Boole, Frege, Russel e Hilbert estudaram muito do que já aparecia na lógica tradicional, como os trabalhos de Aristóteles e Leibniz. Todavia, enquanto a lógica tradicional era considerada parte da filosofia, a lógica moderna foi retirada do domínio filosófico e passou a desempenhar um papel central na matemática. Ela evoluiu para um conceito mais restrito, cuja abordagem

[1] Thomas Hobbes recebeu notoriedade por escrever textos de filosofia voltados para política, em defesa do absolutismo, porém sua formação original foi matemática.

é mais formal do que filosófica. Por esse motivo, essa nova abordagem da lógica foi denominada então: *Lógica Formal*.

O principal objetivo da abordagem formal foi fundamentar, através de conceitos estritamente sólidos, a própria matemática e em especial a *aritmética*. Frege, por exemplo, acreditava que a matemática pudesse ser reduzida à lógica, o que atualmente é conhecido como *logicismo*. Russell, por sua vez, se preocupou em refinar o sistema de Frege, e obteve o que foi o predecessor do atual *cálculo de predicados*. Já Hilbert buscou uma formalização de toda a matemática através de um conjunto restrito de axiomas. Essa nova proposta ficou conhecida como o *Programa de Hilbert* e foi apresentada no *International Congress of Mathematicians* no ano de 1900 em Paris.

No final da década de 50 e início da década de 60, com o surgimento do computador, matemáticos despertaram seu interesse na elaboração de provas mecânicas, ou seja, aquelas realizadas inteiramente dentro e através de procedimentos de uma máquina. Um pioneiro nesse domínio foi *de Bruijn*, através de seu projeto denominado Automath [3]. O nome é sugestivo, pois havia a esperança de que máquinas gerassem provas de teoremas de forma totalmente automática. Hoje, já se sabe que tal ambição é demasiada, e provas mecânicas são utilizadas para certificar as provas realizadas em papel e lápis, e não para automatizar todo o processo de demonstração. Mais além, essa nova área de trabalho ficou conhecida como *Verificação Formal*.

A área de Verificação Formal se estabeleceu então como um importante fragmento da área de Computação, com aplicações em Teoria da Computação e Engenharia de Software. No entanto, o processo de formalização de uma teoria é bem mais complexo do que possa parecer inicialmente. De fato, as provas construídas em papel e lápis possuem abreviações que precisam ser eliminadas para viabilizar sua escrita em uma linguagem puramente formal. Dessa maneira, provas em computador se baseiam menos em intuições, que podem eventualmente se mostrar incorretas, e mais em axiomas e regras de dedução.

Para atender a esse fim, houve a criação de uma nova família de sistemas, os quais viabilizavam a especificação formal de teorias. A ideia fundamental desses programas é considerar um conjunto bastante restrito de regras de dedução e a partir dele construir teorias, que armazenadas em bibliotecas, permitem a elaboração de provas mais complexas, porém menos extensas. Esses sistemas receberam o nome de *Assistentes de Prova* e atualmente, no meio acadêmico e na indústria, existe muito trabalho em provas mecânicas e construção de programas certificados, com auxílio dessa categoria de *software*. Alguns exemplos desses sistemas são dados por: *Coq* [2], *Isabelle/HOL* [4], *PVS* [5] e *Mizar* [6].

Em particular o Coq é uma ferramenta refinada, baseada em três linguagens bem coordenadas e complexas. A primeira é uma linguagem de especificação, chamada *Gallina*. A segunda, chamada *Vernacular*, é uma linguagem de comandos. Ela permite ao usuário orientar o Coq na organização das provas e programas. A terceira linguagem é composta pelo conjunto de *táticas*, as quais possuem construtores lógicos de alto nível que são utilizados para se transformar, de maneira "semi-automática', provas complexas em provas equivalentes e mais simples. De fato, o uso articulado e inteligente dos comandos e táticas permitem a verificação de provas complexas. Assim, a formalização em um assistente de prova, como o Coq, é mais do que simplesmente preencher os detalhes. Na verdade esse é um trabalho desafiador e exige criatividade [7].

O sistema Coq é baseado em uma lógica de ordem superior, que tem como fundamento teórico o *Cálculo de Construções Indutivas (CCI)* [2], o qual por sua vez é uma extensão de um formalismo denominado *cálculo* λ [8]. Esse formalismo foi o produto de uma teoria desenvolvida por A. Church [9] na década de 30, e nada mais é, que um sistema particular de reescrita, o qual se tornou muito popular por ser um modelo teórico de computação equivalente às *máquinas de Turing*, ou seja, é capaz de expressar qualquer função computável. O cálculo λ possui uma única regra de rescrita, chamada β dada por:

(β) $(\lambda x.a)\, b \rightarrow a\{x/b\}$, onde o termo do lado direito da regra representa a *substituição simultânea* de todas as ocorrências livres da variável x em a, pelo termo b.

A definição formal do cálculo considera a operação de substituição como uma meta-operação, isto é, é necessária a definição de um conjunto de regras exterior à gramática do cálculo para que se realize a operação de substituição. Tal dificuldade motivou pesquisas com modificações do cálculo λ, pela adição de elementos em sua gramática e refinamento de suas regras. Nesse contexto foram desenvolvidos sistemas que internalizam a operação de substituição e os cálculos resultantes ficaram conhecidos como *cálculos de substituições explícitas* [1]. Nesse trabalho estamos particularmente interessados no cálculo de substituições explícitas λex [10], porque esse possui uma notação relativamente simples e no entanto satisfaz simultaneamente importantes propriedades como a confluência em termos fechados e abertos, a composição completa, a preservação da normalização forte e a simulação de um passo de β-redução.

A investigação de quais termos de um cálculo são fortemente normalizáveis desempenha um papel particularmente importante, pois tais objetos não possuem caminhos de redução de comprimento infinito. Além disso, pode-se realizar um paralelo entre termos e *programas*; os termos fortemente normalizáveis correspondem exatamente aqueles programas que não consomem um número infinito de passos de execução. Um exemplo simples, de um programa que consome um número infinitamente enumerável de passos, pode ser dado por aquele realiza sequencialmente a impressão de todos os números naturais em uma lista (1, 2, 3, 4, ...). Em cada passo, ele imprime o número seguinte da lista, e dado que o conjunto $\mathbb{N}$ é enumerável e infinito, o programa segue indefinidamente sem um ponto de parada.

De forma que muita atenção tem sido dada à classificação das estratégias de redução no cálculo λ [11, 12, 13, 14, 15, 16, 17]. E assim, uma estratégia de redução do cálculo pode ser classificada como *perpétua* [18], se ela computa para um termo, um caminho de redução de comprimento infinito, caso ele exista; senão ela retorna algum caminho de redução de comprimento $n \in \mathbb{N}$ até uma forma normal.

Em [19], Delia Kesner apresenta uma caracterização indutiva dos termos do λex que são fortemente normalizáveis, através da definição de uma estratégia de redução perpétua para o próprio λex. As demonstrações realizadas não fazem uso de argumentos da lógica clássica, como lei do terceiro excluído ou provas por contradição; e portanto podem ser chamadas de provas construtivas, nos moldes do que é esperado para execução de uma verificação formal, em um assistente como o Coq.

Inicialmente realizamos uma definição indutiva do cálculo λex que não levava em conta as operações de α-conversão, isto é, o renomeamento de variáveis ligadas de um termo.

Essa operação é especialmente necessária para se evitar a captura de variáveis livres, na construção de funções recursivas sobre termos. O principal exemplo de ocorrência dessa questão é a definição recursiva da operação de substituição, que é gerada na regra β. Essa função deveria ser definida para a totalidade dos termos da gramática, porém no caso em que temos de executar $(\lambda y.a)\{x/b\}$, é necessário que se observe duas condições, antes da propagação da função no interior do termo:

(i) $x \neq y$;

(ii) y não pode ocorrer livre em b.

Isso coloca restrições sobre a aplicação da função, e assim a solução comumente dada é considerar a recursividade não na totalidade dos termos, mas sim nas classes de equivalência geradas pela relação $=_\alpha$, onde $(\lambda y.a) =_\alpha (\lambda z.a')$ para todo z que não ocorra livre em a; e a' é igual ao renomeamento de todas as ocorrências livres de y em a, por z.

Nos anos 20 e 30, Frenkel e Mostowski criaram seu *modelo de permutação* com o objetivo de provar a independência do axioma de escolha (AC) e outros axiomas da teoria de conjuntos. Esse trabalho foi posteriormente utilizado como base para definição de uma nova teoria chamada *Lógica Nominal* [20, 21, 22], a qual oferece o subsídio necessário para se trabalhar com nomes ligados e α-equivalência. Ela busca considerar construtores e propriedades que são invariantes com respeito a permutação de nomes, o que dá origem a uma atrativa e simples formalização do comum, mas frequentemente incorreto, uso de recursão e indução para sintaxes abstratas módulo α-equivalência.

Há uma formalização em Coq do cálculo λ, a qual foi desenvolvida em um trabalho de B. Aydemir *et al* [23], e que utiliza lógica nominal para efetuação de α-indução e α-recursão estrutural [24]. No presente trabalho apresentamos uma extensão dessa especificação, a qual inclui a substituição explícita ($s[x/t]$) na gramática de termos. Avançamos também na definição dos sistemas de reescrita, relações de redução, fechos transitivos / reflexivos-transitivos do λex, utilizando para tal fim a biblioteca *sur_les_relations*, desenvolvida em [25]. E mais além, fizemos uso também da biblioteca *CoLoR* [26, 27] para se definir *caminhos de redução* e o predicado SN (Fortemente Normalizável).

Concluímos por demonstrar os primeiros resultados de [19], isto é: um resultado sobre o não crescimento do número de variáveis livres durante o processo de redução; resultados sobre as reduções dos argumentos dentro e fora de uma *meta-substituição*; o lema da *Composição Completa*; o lema da *Simulação de um Passo da β-Redução*; e um primeiro lema anterior ao *Teorema da Perpetualidade*. Todos esses resultados, caminham para obtenção da demonstração do teorema da PSN (Preservação da Normalização Forte) para λ-termos, ou seja, a demonstração de que todo termo fortemente normalizável no cálculo λ, também o é no cálculo λex.

A organização do código fonte e a disposição lógica de todos os arquivos é dada pela Figura 1.1[2]:

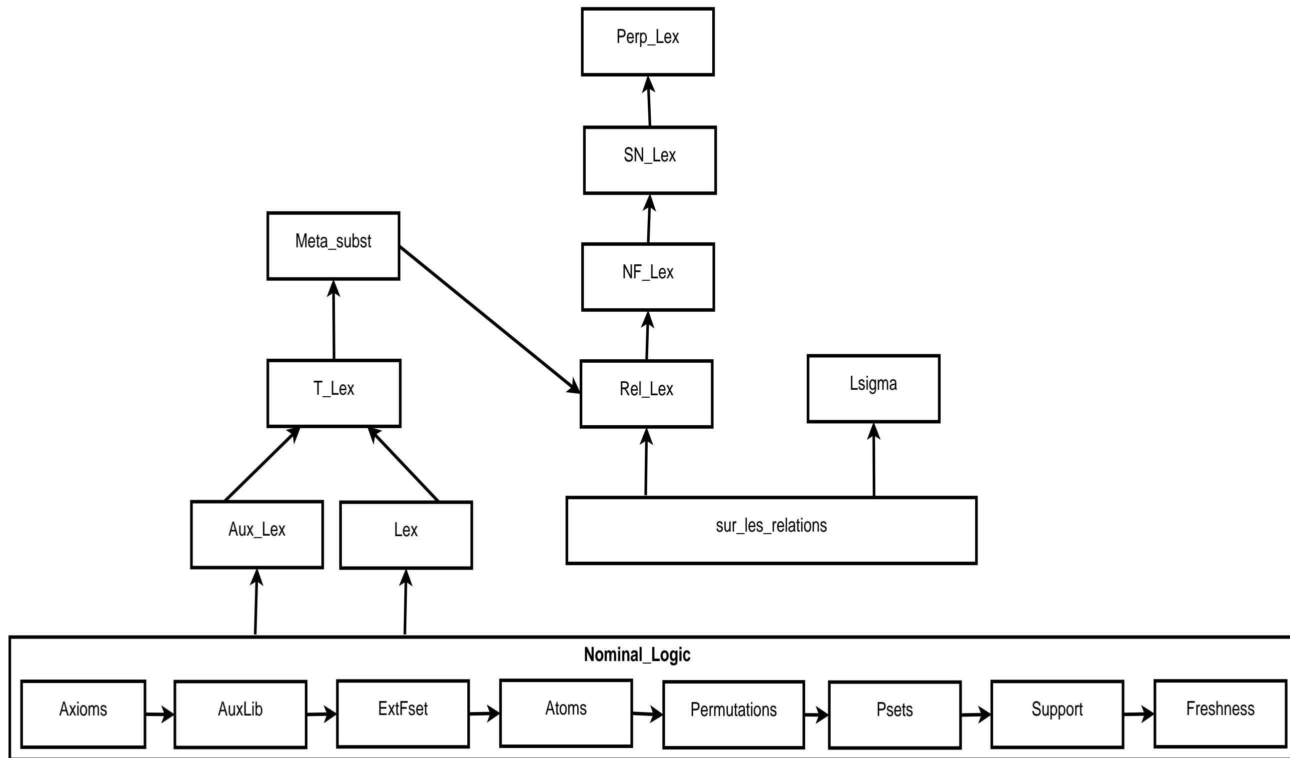


Figura 1.1: *Organização do código fonte.*

(i) `Nominal_logic` é composto de todas as estruturas relativas a lógica nominal, o que é um resultado do agrupamento dos blocos lógicos `Axioms`, `AuxLib`, `ExtFset`, `Atoms`, `Permutations`, `Psets`, `Support` e `Freshness`, encontrados na especificação [23];

(ii) `Lex` e `Aux_Lex`, são respectivamente a especificação da gramática do cálculo λex [10] e uma gramática auxiliar ao cálculo;

(iii) `T_Lex` é a definição de translação entre as gramática λex e λex';

(iv) `Meta_subst` define a função recursiva de *meta-substituição* para o λex;

(v) `sur_les_relations` trata de uma biblioteca de propriedades sobre relações binárias, encontrada em [25];

(vi) `LSigma` é uma especificação adicional, do cálculo $\lambda\sigma$ [1];

(vii) `Rel_Lex` define as *relações de redução* do cálculo λex;

(viii) `NF_Lex` fornece uma definição indutiva para as *formas normais* do λex;

(ix) `SN_Lex` exibe o predicado *fortemente normalizável* no contexto do λex;

(x) `Perp_Lex` contém, for fim, as formalizações obtidas sobre as propriedades do cálculo λex.

[2]Disponibilizamos nosso código fonte, no endereço `http://www.cic.unb.br/~flavio/msc/lex.tar.gz`

Capítulo 2

Fundamentos

Intitulamos essa primeira seção por *Fundamentos* porque, de fato, os conceitos aqui abordados serão utilizados ao longo de todas as demais seções do texto. A primeira subseção tem como objetivo fornecer uma rápida introdução ao funcionamento do sistema Coq [2]; a seguinte subseção inicia uma discussão sobre a definição de sintaxes abstratas através de *termos*; a terceira aborda os fundamentos e propriedades de *Sistemas de Reescrita de Termos*; e concluímos esse capítulo com uma breve apresentação da *Lógica Nominal*, que inclui trechos do código do arquivo `Nominal_Logic` da especificação.

2.1 O sistema Coq

O Coq [2] é uma ferramenta de computação para verificação de provas de teoremas e certificação de programas, cuja confiança reside nas propriedades do Cálculo no qual se baseia, que é chamado de *Cálculo de Construções Indutivas (CCI)*. Esse, é um formalismo que combina muitos dos recentes avanços em lógica, do ponto de vista do cálculo λ [8] e teoria de tipos [28].

A linguagem do Coq é extremamente poderosa e expressiva, tanto para verificação formal quanto para programação lógica. Essa ferramenta tem como finalidade o desenvolvimento de programas para os quais uma confiança absoluta é necessária, como: telecomunicações, transportes, energia, transações bancárias, entre outros. Nesse domínio, a necessidade de programas que atendam rigorosamente suas especificações, justifica o esforço necessário para a verificação formal dos mesmos, visando a criação de códigos seguros para aplicações em condições críticas.

Mas o Coq não é somente útil para o desenvolvimento de programas. Ele também atende a um público de matemáticos que busca formalizar teorias que envolvam provas longas. Suas demostrações são efetuadas de uma maneira interativa e, quando possível, com o auxílio de ferramentas de aplicação automática. Existe bastante trabalho já estabelecido em Coq sobre um grande número de teorias, o qual pode ser encontrado em suas bibliotecas padrão ou em contribuições diversas, como no *site* de *Contribuições de Usuários do Coq* [29].

O processo de realização de uma prova ou especificação de um programa é baseado em um restrito conjunto de regras de dedução. Cada enunciado ou programa é equivalente a um termo do CCI, o qual possui um *tipo*, e a realização da prova ou especificação, corresponde ao problema de *habitação* desse tipo, ou seja, a construção de um termo que possui o tipo dado, tipo dado à partir do conjunto de regras de inferência de tipos do CCI. Seguem três das mais básicas delas, que correspondem à subteoria do cálculo λ simplesmente tipado:

$$\textsc{Var}\ \frac{(x:A)\in\Gamma}{\Gamma\vdash x:A} \qquad \textsc{Lam}\ \frac{\Gamma :: (x:A)\vdash t:B}{\Gamma\vdash(\lambda_{(x:A)}.\,t)\ :\forall(x:A),\,B}$$

$$\textsc{App}\ \frac{\Gamma\vdash e_1 : A\to B \qquad \Gamma\vdash e_2 : A}{\Gamma\vdash e_1 e_2 : B}$$

Figura 2.1: *Cálculo λ simplesmente tipado.*

Na regra **Lam** da Figura acima, $\forall(v:A),\,B$ é equivalente $A\to B$, se B não possui ocorrências livres de v.

Exemplo 2.1.1. *Uma demostração simples, do fato de que:*
$\forall(A:Set)(a\,b:A)(R:A\to A\to Prop),(\forall ab,\,R\,a\,b)\Rightarrow(\forall ab,\,R\,b\,a)$*, pode ser obtida através da aplicação desse pequeno conjunto de regras, e esse trabalho é exibido a seguir:*

$$\Gamma\equiv\Gamma' :: (H:\forall ab:A,\,R\,a\,b) :: (a:A) :: (b:A)$$

$$\textsc{Lam}\ \cfrac{\textsc{App}\ \cfrac{\textsc{App}\ \cfrac{\textsc{Var}\ \frac{(H:\forall ab:A, R\,a\,b)\in\ \Gamma}{\Gamma\vdash\ (H:\forall ab:A, R\,a\,b)} \qquad \textsc{Var}\ \frac{(b:A)\in\ \Gamma}{\Gamma\vdash(b:A)}}{\Gamma\vdash(H\,b):(\forall c:A, R\,b\,c)} \qquad \textsc{Var}\ \frac{(a:A)\in\Gamma}{\Gamma\vdash(a:A)}}{\Gamma\vdash(H\,b\,a):(R\,b\,a)}}{\textsc{Lam}\ \cfrac{\textsc{Lam}\ \cfrac{\Gamma' :: (H:\forall ab:A, R\,a\,b) :: (a:A)\vdash(\lambda_{(b:A)}.H\,b\,a):(\forall b:A, R\,b\,a)}{\Gamma' :: (H:\forall ab:A, R\,a\,b)\vdash(\lambda_{(a\,b:A)}.H\,b\,a):(\forall ab:A, R\,b\,a)}}{\Gamma'\vdash(\lambda_{(H:\forall ab:A, R\,a\,b)(a\,b:A)}\ H\,b\,a):((\forall ab:A, R\,a\,b)\to\forall ab:A, R\,b\,a)}}$$

A aplicação de regras de inferência pode se tornar uma tarefa bastante complexa. Por esse e outros motivos, a linguagem de comandos do Coq agrega um tipo de conceito que é chamado de *tática*. Comandos dessa linguagem permitem aplicar um conjunto de regras, no qual o casamento, entre regra e objetivo, é encontrado de maneira quase que automática. Isso facilita muito o trabalho do usuário, basta ver que na prática, a prova do lema do exemplo 2.1.1 pode ser efetuada conforme exibe a Figura 2.2, e essa tem somente duas linhas de código: "`intros H a b.`" e "`apply H.`"; o que significa introdução seguida de aplicação das hipóteses. Durante a realização de uma prova, o ambiente interativo do Coq exibe uma janela de diálogo que possui uma barra horizontal dupla. As afirmações acima da barra são o conjunto de hipóteses, e abaixo são objetivos de prova. O número

de objetivos e o índice do objetivo corrente são exibidos ao lado direito da barra vertical em `(n/k)`, onde `n` representa o índice do objetivo corrente, e `k` é número de sub-objetivos gerados em cada etapa da prova.

Comandos	**Respostas do Coq**
`Variable A : Set.` `Variable R : A->A->Prop.`	`Variable A declared` `Variable R declared`

`Lemma all_perm:`
`(forall (a b:A), R a b)->`
`(forall (a b:A), R b a).`
`Proof.`
	`======================================= (1 / 1)`
	`(forall (a b:A), R a b)-> forall (a b:A), R b a`

`intros H a b.`	`H : forall (a b:A), R a b`
	`a : A`
	`b : A`
	`======================================= (1 / 1)`
	`R b a`

`apply H.` `Qed.`	`Proof Completed`

Figura 2.2: *Demonstração de* `all_perm` *no Coq.*

O desenvolvimento completo de uma teoria envolve basicamente três tipos de construções: definições indutivas, funções recursivas e proposições. Propriedades admitidas sem demostração recebem o nome de *Axioma*, caso contrário recebem a denominação de *Lema*, *Teorema* ou *Proposição*; não existe diferença sintática entre essas nomenclaturas, ficando a cargo do usuário a escolha da utilização de um ou outro. Em geral os resultados mais fortes e conclusivos são chamados de Teorema e os demais de Lema.

O exemplo clássico de definição indutiva é a construção do tipo *nat*, isso é, a definição do conjunto dos números naturais no Coq (Figura 2.3). Sua declaração possui o seguinte significado: `O` é um natural; `S O` é um natural; `S S O` é um natural, etc. (`O` (constante *zero*) e `S` (função *sucessor*) são os construtores do tipo de dados indutivo *nat*). Adota-se também a notação: `O` $\equiv$ `0`; `S O` $\equiv$ `1`; `S S O` $\equiv$ `2`; `S S S O` $\equiv$ `3, ...`

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

Figura 2.3: *Definição indutiva do tipo* nat.

Um outro exemplo de definição indutiva é construção de árvores binárias, cujas folhas e nós são marcados com números naturais. Definimos esse tipo de objeto, juntamente

com uma função recursiva que conta o número de folhas da árvore dada como argumento, e um lema que afirma que o número de folhas em uma árvore é estritamente maior que zero, como mostrado na Figura 2.4.

```
Require Import Arith.

Inductive tree : Set :=
| leaf : nat -> tree
| node : nat -> tree -> tree -> tree.

Fixpoint n_leaf (t:tree) : nat :=
   match t with
    | leaf a => 1
    | node a t1 t2 => (n_leaf t1) + (n_leaf t2)
   end.

Lemma n_leaf_gt_zero: forall (t:tree), (n_leaf t) > 0.
Proof.
  induction t.
  (* t = leaf n *)
  simpl; auto with arith.
  (* t = node n t1 t2 *)
  simpl; auto with arith.
Qed.
```

Figura 2.4: *Tipo indutivo* `tree`*, função recursiva* `n_leaf` *e lema* `n_leaf_gt_zero`*.*

Declaramos o tipo $tree$ (árvore) que pode ser uma $leaf$ (folha) ou um $node$ (nó). A ideia fundamental dessa definição é a de que todo $leaf\,n$, sendo $n \in \mathbb{N}$, é uma $tree$; e além disso todo $(node\,n\,t1\,t2)$ também o é (com $n \in \mathbb{N}$ e $t1$, $t2$ do tipo $tree$). A semântica pretendida é a de que o tipo indutivo $tree$ represente uma árvore binária de números naturais, conforme o exemplo da Figura 2.5.

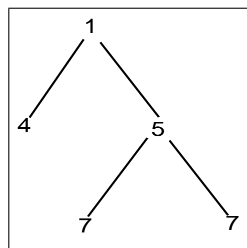


Figura 2.5: *Exemplo de árvore binária de números naturais.*

No final da Figura 2.4, demostramos o Lema `n_leaf_gt_zero` que afirma que, toda árvore tem um número de folhas maior que $zero$. A demostração é bastante simples, e é dada por indução na estrutura de t do tipo $tree$, fazendo uso dos teoremas da biblioteca *Arith* através do comando `auto with arith`, que proporciona uma aplicação automática

de teoremas já demostrados. A biblioteca Arith, é padrão do Coq, e ela foi carregada através do comando `Require Import Arith`.

As demonstrações realizadas em Coq são muitas vezes próximas das provas de papel e lápis, e a vizualização desse tipo de conexão depende do grau de familiaridade que o usuário tem com a ferramenta. No entanto, em algumas situações uma prova em papél não pode ser transcrita de maneira literal para uma prova construtiva, o qual é tipo de demonstração exigida no assistente Coq. Por exemplo, os casos de decidibilidade de uma relação são assumidos "de graça" quando se lida com lógica clássica, mas aqui provar que uma dada relação binária R satisfaz que $R\,a\,b$ ou $\neg R\,a\,b$, para quais quer a e b, exige que se construa uma prova de $R\,a\,b$ ou uma prova de $\neg R\,a\,b$, para todos os casos particulares de a e b. Note que isso só é possível para predicados decidíveis.

2.2 Lógica formal

Iniciaremos com algumas questões e conceitos fundamentais no que se refere à lógica formal. É algo comum, quando se lida com linguagens formais ou lógica matemática, abstrair os detalhes de sintaxe em termos de palavras ou símbolos, através da construção de termos. Isso permite o uso de duas ferramentas inter-relacionadas extremamente úteis: a recursão e as provas por indução sobre a estrutura dos termos. Então seguimos primeiramente com algumas definições.

Definição 2.2.1 (Alfabeto). *Um **alfabeto** $\mathcal{A}$, é definido com um conjunto não vazio de símbolos.*

Exemplo 2.2.2. *São exemplos de alfabetos:*

1. $\mathcal{A} = \{0, 1, 2, ..., 9\}$
2. $\mathcal{A} = \{a, b, c, ..., z\}$
3. $\mathcal{A} = \{\circ, \int, a, d, x, f,), (\}$
4. $\mathcal{A} = \{c_1, c_2, c_3, ...\}$

Toda sequência finita de símbolos de um alfabeto $\mathcal{A}$ é denominada uma *string* ou *palavra*. E qualquer subconjunto, do conjunto de todas as palavras do alfabeto $\mathcal{A}$ (denotado por $\mathcal{A}^*$), é chamada *linguagem*.

Deseja-se construir *linguagens formais* nas quais se possa formular os axiomas, teoremas, e provas. Em tal contexto os conectivos, os quantificadores e a relação de igualdade aparecem com um papel de suma importância. Portanto são incluídos os seguintes símbolos nas chamadas *linguagens de primeira ordem*: $\neg$ (para *"não"*), $\wedge$ (para o *"e"*), $\vee$ (para o *"ou"*), $\rightarrow$ (para o *"se então"*), $\leftrightarrow$ (para o *"se e somente se"*), $\forall$ (para o *"para todo"*), $\exists$ (para o *"existe"*) e $\equiv$ (como o símbolo de igualdade). Segue a definição do alfabeto da *linguagem de primeira ordem*, que é dada como base nesses símbolos:

Definição 2.2.3. *O* ***alfabeto da linguagem de primeira ordem*** *(denotado por $\mathcal{A}$) contém o seguinte conjunto de símbolos:*

(i)	$v_0, v_1, v_2, ...$	[variáveis]
(ii)	$c_0, c_1, c_2, ...$	[constantes]
(iii)	$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$	[conectivos]
(iv)	$\forall, \exists$	[quantificadores]
(v)	$\equiv$	[igualdade]
(vi)	$), ($	[parênteses]

(vii) Para $n \geq 1$:

(1) Símbolos de relações n-árias;

(2) Símbolos de funções n-árias;

As palavras construídas com base em um alfabeto podem, ou não, ter um sentido (por exemplo a palavra $f \wedge c\, v_3 R \vee$ não tem nenhum significado lógico aparente). Logo a construção de *termos* e *fórmulas* é caracterizada pela seguinte definição indutiva:

Definição 2.2.4. ***termos*** *e* ***fórmulas*** *são definidos indutivamente sobre $\mathcal{A}^*$ por:*

(T1) Toda variável é um termo;

(T2) Toda constante é um termo;

(T3) Se $t_1, t_2, ..., t_n$ são termos e f é um símbolo de função n-ária, então $f(t_1, ..., t_n)$ é também um termo.

(F1) Se t_1 e t_2 são termos, então $t_1 \equiv t_2$ é uma fórmula;

(F2) Se as palavras $t_1, t_2, ..., t_n$ são termos e R é um símbolo de relação n-ária, então $R\, t_1 \ldots t_n$ é uma fórmula;

(F3) Se φ é uma fórmula, então $\neg\varphi$ também é uma fórmula;

(F3) Se φ e ψ são fórmulas, então $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ e $(\varphi \leftrightarrow \psi)$ também são fórmulas;

(F4) Se φ é uma fórmula e x é uma variável, então $\forall x\, \varphi$ e $\exists x\, \varphi$ também são fórmulas.

Adicionalmente, a operação de substituição representa uma tarefa básica em sistemas computacionais. Intuitivamente, uma substituição de primeira ordem deverá receber um termo com ocorrências de uma dada variável, e substituir simultâneamente todas as ocorrências dessa por um outro termo.

Definição 2.2.5 (Substituição de primeira ordem)**.** *Uma* ***substituição de primeira ordem*** *θ é um conjunto finito da forma $\{v_1/t_1, ..., v_n/t_n\}$, onde cada v_i é uma variável distinta e cada t_i é um termo distinto de v_i. Cada "v_i/t_i" é denominado uma* ***ligação*** *da substituição θ.*

Definição 2.2.6 (Composição de substituições)**.** *Sejam $\theta = \{u_1/s_1, ..., u_m/s_m\}$ e $\sigma = \{v_1/t_1, ..., v_n/t_n\}$ substituições. A* ***composição*** *$\theta\sigma$, de θ e σ, é a substituição obtida do conjunto:*

$$\{u_1/s_1\sigma, ..., u_m/s_m\sigma, v_1/t_1, ..., v_n/t_n\},$$

Onde deve-se eliminar qualquer ligação $u_i/s_i\sigma$, para a qual $u_i = s_i\sigma$ e qualquer ligação v_j/t_j para a qual $v_j \in \{u_1, ..., u_m\}$. ***Nota:*** *Se t é um termo e σ uma substituição, então $t\sigma$ denota a aplicação da substituição σ ao termo t.*

Definição 2.2.7 (Conjunto de subtermos)**.** *O* ***conjunto de subtermos*** *de um dado termo t, denotadado por $S(t)$, é definido recursivamente por:*

(i) Se t é uma constante ou variável, então $S(t) = t$;

(ii) Se $t \equiv f\, t_1 \,....\, t_n$, então para $i \in \mathbb{N}$:

$$S(t) = \bigcup_{i=1}^{n} \{t_i\} \cup S(t_i)$$

.

Observação: *Se $s \in S(t)$, então s é chamado* ***subtermo*** *de t.*

Definição 2.2.8 (Conjunto de subtermos próprios)**.** *O* ***conjunto de subtermos próprios*** *de um dado termo t, denotado por $S^*(t)$, é definido por: $S^*(t) := S(t) \setminus \{t\}$.*

Observação: *Se $s \in S^*(t)$, então s é chamado* ***subtermo próprio*** *de t.*

Definição 2.2.9 (Posições válidas dos termos)**.** *O conjunto de* ***posições válidas*** *de um termo t, denotado por $O(t)$, é definido recursivamente por:*

(i) Se t é uma constante ou variável, então $O(t) = \{\lambda\}$;

(ii) Se $t \equiv f\, t_1 \,....\, t_n$, então para $i \in \mathbb{N}$:

$$O(t) = \{\lambda\} \cup \bigcup_{i=1}^{n} \{i \cdot \pi \mid \pi \in O(t_i)\}$$

.

Exemplo 2.2.10. *Se $t \equiv f(g(x, y), c)$, então*

$$O(t) = \{\lambda\} \cup \{1 \cdot \lambda\},\ \{1 \cdot 1 \cdot \lambda\} \cup \{1 \cdot 2 \cdot \lambda\} \cup \{2 \cdot \lambda\} = \{\lambda,\ 1 \cdot \lambda,\ 1 \cdot 1 \cdot \lambda,\ 1 \cdot 2 \cdot \lambda,\ 2 \cdot \lambda\}.$$

Notação 2.2.11. *Para $\pi \in O(t)$, o subtermo na posição π de t é denotado por $t|_\pi$, e assim:*

(ii) Para todo t, $\lambda \in O(t)$ e $t|_\lambda \equiv t$;

(iii) $t|_{i \cdot \pi} \equiv t_i|_\pi$.

Exemplo 2.2.12. *Uma observação importante é a de que todo termo pode ser representado por uma* árvore, *tal que: os símbolos de funções são os* nós *e as variáveis ou*

constantes são as folhas*. Por exemplo, o termo* $t \equiv f(g(x,y),c)$ *tem o nó* raiz f*, um nó filho á esquerda* g*, e direita a folha* c*. O nó* g *tem duas folhas* x *e* y*. E nesse caso:*

$$\begin{array}{rcccccccccccc} S(t) & = & \{ & t|_{\lambda} & , & t|_{1\cdot\lambda} & , & t|_{1\cdot 1\cdot\lambda} & , & t|_{1\cdot 2\cdot\lambda} & , & t|_{2\cdot\lambda} & \} \\ & & & ||| & & ||| & & ||| & & ||| & , & ||| & \\ & = & \{ & f(g(x,y),c) & , & g(x,y) & , & x & , & y & , & c & \} \end{array}$$

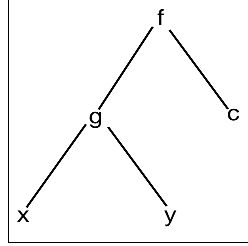


Figura 2.6: *Exemplo de representação em árvore do termo* $f(g(x,y),c)$*.*

Notação 2.2.13. $s[\pi \leftarrow t]$ *denota o termo resultante de substituir na posição* $\pi \in O(s)$*, o termo* $s|_\pi$ *por* t*.*

Definição 2.2.14 (Variáveis livres de um termo)**.** *Seja* t *um termo.* $\mathrm{fv}(t)$ *denota o conjunto de variáveis livres de* t*, que é definido recursivamente através de:*

(i) *Se* $t \equiv c$ *constante, então* $\mathrm{fv}(t) = \emptyset$*;*

(ii) *Se* $t \equiv x$ *variável, então* $\mathrm{fv}(t) = \{x\}$*;*

(iii) *Se* $t \equiv f\, t_1 \ldots . t_n$*, então para* $i \in \mathbb{N}$*:*

$$\mathrm{fv}(t) = \bigcup_{i=1}^{n} \mathrm{fv}(t_i)$$

2.3 Sistemas de reescrita

As equações são ferramentas de grande relevância na matemática e nas ciências como um todo. Os *Sistemas de Reescrita* [30], nada mais são, que restrições para aplicação de um conjunto de equações. Isto é, as igualdades são substituídas por regras de redução, as quais são orientadas para se computar a forma mais *simples* possível de um dado objeto.

Definição 2.3.1 (Sistema de reescrita)**.** *Seja* M *um conjunto e* $\rightarrow$ *uma relação binária sobre* M*.* $(M, \rightarrow)$ *é um* ***sistema de reescrita****.* $\rightarrow$ *é denominada a relação de redução ou relação de reescrita correspondente ao sistema.*

Considere um sistema de reescrita $(M, \rightarrow)$

(i) $\leftarrow$ *denota a relação inversa de* $\rightarrow$

(ii) $\leftrightarrow = \rightarrow \cup \leftarrow$

(iii) $\xrightarrow{n}$ *é definida indutivamente como segue:*

(1) $u \xrightarrow{0} v$ *sse* $u \equiv v$

(2) $u \xrightarrow{n+1} v$ *sse* $\exists w, u \xrightarrow{n} w \to v$

(iv) $\xrightarrow{+} = \bigcup_{n\geq 1} \xrightarrow{n}$ *é o* ***fecho transitivo*** *de* $\to$

(v) $\xrightarrow{*} = \bigcup_{n\geq 0} \xrightarrow{n}$ *é o* ***fecho reflexivo-transitivo*** *de* $\to$

```
Variable A : Set.

Inductive explicit_rel_plus (R : A -> A -> Prop) :  A -> A -> Prop :=
 | relplus_1step :
  forall x y :  A, R x y -> explicit_rel_plus R x y
 | relplus_trans1 :
  forall x y z :  A, R x y ->
  explicit_rel_plus R y z -> explicit_rel_plus R x z.
```

Figura 2.7: *Fecho transitivo de uma relação binária.*

```
Variable A : Set.

Inductive explicit_star (R : A -> A -> Prop) :  A -> A -> Prop :=
 | star_refl :  forall x :  A, explicit_star R x x
 | star_trans1 :
  forall x y z :  A, R x y ->   explicit_star R y z -> explicit_star R x z.
```

Figura 2.8: *Fecho reflexivo-transitivo de uma relação binária.*

No arquivo `sur_les_relations` [25] encontramos definições indutivas para os fechos transitivo ($\xrightarrow{+}$) e reflexivo-transitivo ($\xrightarrow{*}$) de uma relação binária `R`, conforme é apresentado respectivamente nas Figuras 2.7 e 2.8. Mas elas não são a transcrição literal da Definição 2.3.1. De fato, a definição indutiva de fecho-reflexivo na Figura 2.7 afirma que: $\forall xy$, se $x \to y$, então $x \xrightarrow{+} y$; e além disso $\forall xyz$, se $x \to y$ e $y \xrightarrow{+} z$, então $x \xrightarrow{+} z$. Contudo essa caracterização indutiva acaba por ser logicamente equivalente ao que é apresentado na Definição 2.3.1. O mesmo ocorre com a definição do fecho reflexivo-transitivo.

Se M é o conjunto de termos da Definição 2.2.4, então é possível definir um sistema de reescrita para tal conjunto e o chamamos *Sistema de Reescrita de Termos*.

Definição 2.3.2. *Seja* M *um conjunto de termos:*

(i) Uma ***regra*** $l \to r$ *é um par ordenado de termos* $l, r \in M$*, sendo que* l *não é uma variável e* $\mathrm{fv}(r) \subseteq \mathrm{fv}(l)$*;*

*(ii) Um **sistema de reescrita de termos**, abreviado por TRS (*Term Rewriting System*) é um conjunto M de termos com um conjunto de regras $\mathcal{R}$;*

*(iii) Dado um TRS, o conjunto de regras $\mathcal{R}$ define a **relação de redução de termos**, denota por $\rightarrow_{\mathcal{R}}$ sobre M da seguinte forma;*
Para todo t_1, t_2, $t_1 \rightarrow_{\mathcal{R}} t_2$, se e somente se, existem:

(1) $l \rightarrow r \in \mathcal{R}$,

(2) $\pi \in O(t_1)$ e σ uma substituição com: $t_1|_\pi \equiv l\sigma$ e $t_2 \equiv t_1[\pi \leftarrow r\sigma]$

Dado um Sistema de Reescrita, alguns conceitos importantes podem ser definidos. Se um elemento de um sistema de reescrita não possui nenhum caminho infinito de redução ele é chamado *fortemente normalizável*; quando ao final de uma sequência de aplicações de regras obtemos um termo t, que não pode mais ser reduzido, dizemos que t é uma *forma normal*.

E se para qualquer termo, a ordem de aplicação das regras do sistema não é relevante, pois a forma normal resultante, caso exista é única, este sistema será denominado *confluente*. Seguem portanto as definições formais desses conceitos, juntamente com exemplos das definições de *forma normal* e *fortemente normalizável* em Coq.

Definição 2.3.3. *Se $(M, \rightarrow)$ é uma sistema de reescrita, $u, v \in M$ são denominados **juntáveis** se existe um $w \in M$ tal que $u \xrightarrow{*} w \xleftarrow{*} v$. Para denotar que dois termos u, v são juntáveis usar-se-á a seguinte notação: $u \downarrow v$*

Definição 2.3.4 (Confluência)**.** *Um sistema de reescrita $(M, \rightarrow)$ é **confluente** sse, para todo $u, v, w \in M$ com $v \xleftarrow{*} u \xrightarrow{*} w$, existe algum $r \in M$ com $v \xrightarrow{*} r \xleftarrow{*} w$ (ou simplesmente $v \downarrow w$).*

Definição 2.3.5 (Forma Normal)**.** *Um elemento u é dito forma normal com respeito a $\rightarrow_{\mathcal{R}}$, se para todo v **não** ocorre que $u \rightarrow_{\mathcal{R}} v$. Denota-se $v \in \mathcal{NF}_{\mathcal{R}}$.*

Em Coq definimos `NF` como uma tradução literal da Definição 2.3.5:

```
Definition NF (R : tm -> tm -> Prop) (s :  tm) := forall t, ~ R s t.
```

Figura 2.9: *Forma normal.*

Definição 2.3.6 (Caminho de Redução)**.** *Um Caminho de Redução é uma sequência de termos, u_i, com $i \in \mathbb{N}$. De forma que: $u_i \rightarrow u_{i+1}$.*

Definição 2.3.7 (Comprimento de um Caminho de Redução)**.** *Se um caminho de redução tem um número finitamente contável de termos $n \in \mathbb{N}$, então seu comprimento é $n-1$; Caso contrário seu comprimento é ∞.*

Definição 2.3.8 (Normalização fraca)**.** *Um elemento $u \in M$ é **fracamente normalizável** se existe pelo menos um caminho de redução a partir de u que tem comprimento diferente de ∞. Denota-se: $u \in \mathcal{WN}_{\mathcal{R}}$.*

Definição 2.3.9 (Normalização forte)**.** *Um elemento $u \in M$ é* ***fortemente normalizável*** *se qualquer caminho de redução a partir de u tem comprimento diferente de ∞. Denota-se: $u \in \mathcal{SN}_{\mathcal{R}}$.*

Alternativamente, pode-se definir a noção de normalização forte de forma construtiva. Essa definição em Coq é feita indutivamente como a seguir:

```
Inductive SN (R : tm -> tm -> Prop) :  tm -> Prop :=
  | reg_SN : forall t, (forall s, R t s -> SN R s) -> SN R t.
```

A definição indutiva é mais amplamente utilizada em verificações formais [26, 27] e provas construtivas [19], que a Definição 2.3.9.

Definição 2.3.10 (Comprimento Máximo de um Caminho de Redução)**.** *Se $u \in \mathcal{SN}_{\mathcal{R}}$, é possível definir o comprimento máximo de um caminho de redução à partir de u, o qual é denotado por: $l_{\mathcal{R}}(u) = max\{n \,|\, \exists u' \in \mathcal{NF}_{\mathcal{R}} : u \xrightarrow{n}_{\mathcal{R}} u'\}$*

Utilizamos as definições 2.3.3 a 2.3.10 nas demonstrações de propriedades do cálculo λ [8], como por exemplo no Teorema 3.4.5, e também em verificações das extensões do cálculo λ. Em especial abordamos as características de normalização do λex [10], e assim as Definições 2.3.5, 2.3.6, 2.3.7, 2.3.9 e 2.3.10 exercem papel fundamental, nas formalizações apresentadas na Seção 5.3.

2.4 Lógica nominal

A abordagem convencional da lógica formal não é suficientemente para expressar linguagens que envolvem construtores com variáveis ligadas. Uma afirmação comumente realizada é a de que, a coleção de termos pode ser representada através de um conjunto quociente, módulo uma relação de α-equivalência conveniente. Mas como fazer boas escolhas de representantes dessas classes de α-equivalência?

A ”Convenção de Variáveis de Barendregt” [8] afirma que a escolha de um representante deve obedecer o seguinte princípio: sempre as variáveis ligadas devem ter nomes novos, isto é, mutualmente distintos de qualquer outro nome de variável (livre) no contexto corrente. Contudo, apesar de ser comumente afirmado que a convenção de Variáveis de Barendregt permite trabalhar com classes de α-equivalência sobre termos, isso não é correto quando se lida com recursão ou indução estrutural.

Exemplo 2.4.1. *Quando realizamos uma prova de uma dada propriedade P, através de simples indução sobre a estrutura de λ-termos, devemos provar que: $P(a) \to P(\lambda x.a)$; para toda variável x e todo λ-termo a. Nesse caso, não é possível colocar restrições sobre o conjunto ao qual x pertence. Um caso concreto desse problema ocorre quando tentamos demostrar, por exempo, que para todo t, y, $u \in \Lambda$:*

$$y \notin \text{fv}(t) \Rightarrow t\{y/u\} \equiv t.$$

De fato, se realizarmos indução estrural ordinária, isso é, aquela que não considera classes de α-equivalência, teremos que abordar o caso em que $t \equiv \lambda x.s$:

$$y \notin \mathrm{fv}(\lambda x.s) \Rightarrow (\lambda x.s)\{y/u\} \equiv \lambda x.s.$$

Mas, para que possamos propagar a meta-substituição $\{y/u\}$ em $(\lambda x.s)$ e fazer uso da hipótese de indução: $y \notin \mathrm{fv}(s) \Rightarrow s\{y/u\} \equiv s$, necessitamos da garantia de que $x \notin \mathrm{fv}(u)$ (pela construção da função recursiva de meta-substituição), e isso é um impedimento para conclusão da demonstração, dado que não consiguimos deduzir tal informação do nosso conjunto de hipóteses.

A abordagem *nominal* de sintaxes abstratas oferece o subsídio necessário para se trabalhar com teorias que possuem ligações de nomes (*binding*) e α-equivalência. Ela busca considerar construtores e propriedades que são invariantes com respeito a permutação de nomes, o que dá origem a uma atrativa e simples formalização uso de recursão e indução para sintaxes abstratas módulo α-equivalência [24]. Nosso objetivo aqui é justamente obter uma formalização dos princípios de indução e recursão sobre classes de α-equivalência, porém não varremos todos os detalhes dessa teoria. Essa seção busca fornecer um resumo dos principais conceitos que resultam nos teoremas de indução e recursão aplicados a gramática do cálculo λ em [23], o qual buscamos estender para o λex.

Os princípios usuais de indução e recursão estrutural são parametrizados por uma assinatura algébrica que especifica quais são os possíveis construtores dos termos. Necessitamos então fixar uma noção de assinatura que também especifique as ligações de variáveis que ocorram nos termos, aqui utilizaremos a noção de *assinatura nominal* [24]. Do ponto de vista formal as ligações de variáveis precisam ser atômicas, no sentido de que suas estruturas são imateriais, se comparadas por um critério de distinção entre dois nomes de mesmo tipo. Portanto, essa classe de objetos é chamada de *átomo*, em uma assinatura nominal.

Serão definidos dois conjuntos: o conjunto $\mathbb{A}$ de todos os *átomos* e $\mathbb{AS}$ de todos os *sorts de átomos*. Também se define uma função $sort \in \mathbb{A} \to \mathbb{AS}$, que designa *sorts* para átomos e determina que os conjuntos $\mathbb{AS}$ e $\mathbb{A}_{\mathtt{a}} := \{a \in \mathbb{A} \mid sort(a) = \mathtt{a}\}$ sejam enumeráveis. Uma *assinatura nominal* Σ consiste então de um subconjunto de *sorts* de átomos, $\Sigma_{\mathtt{A}} \subseteq \mathbb{AS}$, um conjunto $\Sigma_{\mathtt{D}}$ de *sorts de dados* e um conjunto $\Sigma_{\mathtt{C}}$ de *construtores*. Cada construtor $K \in \Sigma_{\mathtt{C}}$ tem uma *aridade* σ e um *sort de resultado* $\mathtt{s} \in \Sigma_{\mathtt{D}}$, o que é notacionado por $K : \sigma \to \mathtt{s}$.

Na especificação de [23] a definição de átomo utiliza um recurso de definição indutiva chamado de `Record`, o qual permite uma maleabilidade maior nas declarações sobre o tipo em questão. Esse artifício é utilizado pela primeira vez na especificação na definição do tipo `ExtFset`, que modela o comportamento de *conjuntos extencionais finitos*. Os tipos `aset` e `AtomT` são construídos sobre a definição de `ExtFset`, para que o primeiro represente um conjunto finito de átomos e o segundo, um conjunto infinito de objetos dessa mesma classe.

Definição 2.4.2 (Aridade)**.** *O Conjunto de* ***aridades*** *é obtido pela seguinte definição indutiva:*

(i) Todo sort *de átomo* $\mathtt{a} \in \Sigma_{\mathtt{A}}$ *é uma aridade;*

(ii) Todo sort *de dados* $\mathtt{s} \in \Sigma_{\mathtt{D}}$ *é uma aridade;*

(iii) A aridade unitária $\mathtt{1}$ *é uma aridade;*

(iv) Se σ_1 *e* σ_2 *são aridades, então* $\sigma_1 * \sigma_2$ *é uma aridade;*

(v) Se $\mathtt{a} \in \Sigma_{\mathtt{A}}$ *e* σ *é uma aridade, então* $\ll\!\mathtt{a}\!\gg\sigma$ *é uma aridade.*

```
Record AtomT : Type := mkAtom {
 atom :  Set;
 asetR : ExtFset atom;
 aset := extFset asetR;
 atom_eqdec :  forall (a b :  atom), {a = b} + {a <> b};
 atom_infinite :  forall (S : aset), { a :  atom | ~ In a S }
} .
```

Figura 2.10: *Átomos.*

Definição 2.4.3. *O conjunto de* ***termos*** *em uma assinatura nominal* Σ*, com suas respectivas aridades, onde* $t : \sigma$ *indica que* t *tem aridade* σ*, é definido por:*

(i) Se $a \in \mathbb{A}_{\mathtt{a}}$ *é um átomo de* sort $\mathtt{a}$*, então* $a : \mathtt{a}$ *é um termo;*

(ii) Se $K : \sigma \to \mathtt{s}$ *pertence a* $\Sigma_{\mathtt{C}}$ *e* $t : \sigma$ *é um termo, então* $Kt : \mathtt{s}$ *também é um termo;*

(iii) O termo $\langle\,\rangle : \mathtt{1}$ *é o único termo de aridade unitária;*

(iv) Se $t_1 : \sigma_1$ *e* $t_2 : \sigma_2$ *são termos, então* $\langle t_1, t_2 \rangle : \sigma_1 * \sigma_2$ *também é um termo;*

(v) Se $a \in \mathbb{A}_{\mathtt{a}}$ *e* $t : \sigma$ *é um termo, então* $\ll\!a\!\gg t :\ll\!\mathtt{a}\!\gg\sigma$ *também é um termo.*

Denotam-se $Ar(\Sigma)$ o conjunto de todas as aridades sobre a assinatura Σ, $\mathrm{T}(\Sigma)$ o conjunto de todos os termos sobre Σ, e $ar \in \mathrm{T}(\Sigma) \to Ar(\Sigma)$ a função de designação de um termo t para uma única aridade σ. Para cada $\sigma \in Ar(\Sigma)$, denota-se $\mathrm{T}(\Sigma)_\sigma$ o subconjunto $\{t \in \mathrm{T}(\Sigma) \,|\, ar(t) = \sigma\}$, de termos de aridade σ.

Exemplo 2.4.4. *O exemplo clássico de uma assinatura nominal é a definição da gramática do cálculo* λ*:*

Exemplificando:

- *A constante* $\bullet$ *é reresentada por* D*;*
- *A variável* x *é representada por* Vx*;*
- $(a\,b)$ *é represntado por* $A\langle a, b\rangle$*;*
- $\lambda x.(x\,y)$ *é representado por* $L \ll\!x\!\gg A\langle Vx, Vy\rangle$*;*

sorts de átomos	sorts de dados	construtores
$\mathtt{V}$	$\mathtt{t}$	$D : \mathtt{t}$
		$V : \mathtt{v} \to \mathtt{t}$
		$A : \mathtt{t} * \mathtt{t} \to \mathtt{t}$
		$L : \ll \mathtt{v} \gg \mathtt{t} \to \mathtt{t}$

Figura 2.11: *Assinatura nominal do cálculo* λ.

- $\lambda xz.((x\,y)\,z))$ *é representado por* $L \ll x \gg (L \ll z \gg A\langle A\langle Vx, Vy\rangle, Vz\rangle)$.

Os termos sobre a assinatura Σ são justamente as árvores de sintaxe abstrata determinadas sobre a assinatura ordinária (considerada na totalidade dos termos, e não nas classes de α-equivalência) associada a Σ, os quais os *sorts* são as aridades de Σ, os construtores são aqueles de Σ, somados a construtores para a unidade, pares e ligações de átomos, além dos próprios átomos considerados como constantes particulares. Consequentemente pode-se utilizar recursão ordinária estrutural para se definir funções do conjunto $\mathrm{T}(\Sigma)$ para Σ; e também é possível fazer uso de indução ordinária estrutural para se provar propriedades sobre esse termos.

Ainda não foi considerado o fato de que, termos do tipo $\ll a \gg t$ devem ser identificados a menos de renomeamentos do átomo a. Dada uma assinatura nominal Σ, a relação de *α-equivalência*, $t =_\alpha t' : \sigma$ (onde $\sigma \in Ar(\Sigma)$ e $t, t' \in \mathrm{T}(\Sigma)_\sigma$) faz tais identificações, e é indutivamente definida pelas seguintes regras.

Definição 2.4.5 (Definição de α-equivalência em uma assinatura nominal)**.** *Em uma assinatura nominal Σ a relação de α-equivalência entre termos é definida por:*

$$(=_\alpha .1)\ \frac{a \in \Sigma_{\mathtt{A}} \qquad a \in \mathbb{A}_{\mathtt{a}}}{a =_\alpha a : \mathtt{a}} \qquad (=_\alpha .2)\ \frac{(K : \sigma \to \mathtt{s} \in \Sigma_{\mathtt{C}}) \qquad t =_\alpha t' : \sigma}{Kt =_\alpha Kt' : s}$$

$$(=_\alpha .3)\ \frac{}{\langle\,\rangle =_\alpha \langle\,\rangle : \mathtt{1}} \qquad (=_\alpha .4)\ \frac{t_1 =_\alpha t_1' : \sigma_1 \qquad t_2 =_\alpha t_2' : \sigma_2}{\langle t_1, t_2\rangle =_\alpha \langle t_1', t_2'\rangle : \sigma_1 * \sigma_2}$$

$$(=_\alpha .5)\ \frac{\mathtt{a} \in \Sigma_{\mathtt{A}} \quad a, a', a'' \in \mathbb{A}_{\mathtt{a}} \quad a'' \notin atm(\langle a, t, a', t'\rangle) \quad t\{a := a''\} =_\alpha t'\{a' := a''\} : \sigma}{\ll a \gg t =_\alpha \ll a' \gg t' : \ll \mathtt{a} \gg \sigma}$$

Figura 2.12: α*-equivalência.*

Na regra $(=_\alpha .5)$, $atm(t)$ indica o conjunto finito de átomos que ocorre em t; e $t\{a := a'\}$ indica o termo resultante após a substituição de todas as ocorrências de a em t por a' (assumindo que ambos são do mesmo *sort*). Essas duas operações podem ser definidas através de recursão ordinária estrutural.

Definição 2.4.6 (Funções: Conjunto de Átomos e Substituição)**.** *Dado um termo, as seguintes funções* ***conjunto de átomos*** *e* ***substituição*** *(sem prevenção de captura de variáveis), podem ser definidas através de recursão ordinária, respectivamente por:*

$$
\begin{array}{lrcl}
(i) & atm(a) & = & \{a\} \\
(ii) & atm(Kt) & = & atm(t) \\
(iii) & atm(\langle\,\rangle) & = & \emptyset \\
(iv) & atm(\langle t_1, t_2\rangle) & = & atm(t_1) \cup atm(t_2) \\
(v) & atm(\ll a \gg t) & = & \{a\} \cup atm(t)
\end{array}
$$

$$
\begin{array}{lrcl}
(i) & a''\{a := a'\} & = & \begin{cases} a' & \text{se} \quad a'' = a \\ a'' & \text{se} \quad a'' \neq a \end{cases} \\
(ii) & (Kt)\{a := a'\} & = & K(t\{a := a'\}) \\
(iii) & \langle\,\rangle\{a := a'\} & = & \langle\,\rangle \\
(iv) & \langle t_1, t_2\rangle\{a := a'\} & = & \langle t_1\{a := a'\}, t_2\{a := a'\}\rangle \\
(v) & (\ll a'' \gg t)\{a := a'\} & = & \ll a''\{a := a'\} \gg t\{a := a'\}
\end{array}
$$

Figura 2.13: *Funções recursivas: Conjunto de Átomos e Substituição.*

O ingrediente crucial na formulação da indução e recursão estrutural, para α-termos sobre uma assinatura nominal, é conceito de *suporte finito*. Ele fornece uma maneira elegante, baseada em ações de permutações, de se expressar o fato que um dado átomo a é *fresh* (novo) para um objeto matemático. Isso permite caracterizar não somente átomos novos para termos, o que coincide com a definição do conjunto de variáveis que não ocorrem livres no termo, mas também definir o significado de átomo novo para objetos de tipo funcional. Então, consideremos o seguinte conjunto de afirmações:

(i) Seja $Perm$ o conjunto de todas as *permutações (finitas) de átomos*, cujos elementos são, por definição, bijeções $\pi : \mathbb{A} \leftrightarrow \mathbb{A}$ tais, que o conjunto $\{a \in \mathbb{A} \,|\, \pi(a) \neq a\}$ é finito e $sort(\pi(a)) = sort(a)$;

(ii) A operação de composição de permutações $\circ \in (Perm \times Perm \rightarrow Perm)$ é definida por : $(\pi \circ \pi')(a) := \pi(\pi'(a))$;

(iii) $(Perm, \circ)$ é um grupo de permutações, onde ι denota a permutação identidade e π^{-1} representa o inverso de π;

(iv) Dentre os elementos de $Perm$ destacam-se as transposições (ou *swapps*) $(a\,a')$, que nada mais são que um mapeamento de a para a' e de a' para a, fixando os todos os demais átomos;

(v) É um fato básico de teoria de grupos, que qualquer $\pi \in Perm$ é igual a composição de número finito de transposições, assim uma permutação pode ser representada por uma lista finita de transposições. Por exemplo: se $a, a', b, b', c, c' \in \mathbb{A}$ então $[(a\,a'), (b\,b'), (c\,c')]$ é um elemento de $Perm$.

Definição 2.4.7 (Ação de Permutações)**.** *Uma ação de* $Perm$ *no conjunto* X *é um função pertencente a* $(Perm \times X \to X)$*, cujo efeito sobre* $(\pi, x) \in Perm \times X$ *é denotado por* $\pi.x$*, onde são observadas as seguintes propriedades, para todo* $x \in X$ *e* $\pi, \pi' \in Perm$*:*

(i) $\iota.x = x$

(ii) $\pi.(\pi'.x) = (\pi \circ \pi').x$

```
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.

Record SwapT (A : AtomT) (X : Set) : Set := mkSwap {
 swap : (A * A) -> X -> X;
 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 (a, b) c, swapa A (a, b) d) (swap (a, b) x) }.
```

Figura 2.14: *Transposições ou Swaps.*

```
Record PsetT (A : AtomT) (X : Set) : Set := mkPset {
 perm : permt A -> X -> X;
 perm_id : forall x, perm [] x = x;
 perm_compose : forall p q x, perm (p :++ q) x = perm p (perm q x)
 }.
```

Figura 2.15: *Ações de permutações.*

Em [23] ações de permutações (Figura 2.15) são definidas pelo `Record PsetT`. *Swaps* são definidos primeiro para átomos, como uma função que recebe um par ordenado de elementos e um terceiro elemento a ser trocado. Se o terceiro elemento for igual a qualquer um dos outros dois, no par, então ele será trocado, caso contrário nada é realizado. A ação de um *swap* sobre um objeto qualquer do tipo `Set` é definida através do `Record SwapT`.

Definição 2.4.8 (Suporte e Conjunto Nominal)**.** *Dada uma ação de* $Perm$*, um conjunto* X*, e um elemento* $x \in X$*:*

(i) Sendo $A \subseteq \mathbb{A}$ *um conjunto de átomos, então se para todo* $a, a' \notin A$ *de mesmo* sort *e* $\pi = [(a\, a')] \in Perm$*, tem-se* $\pi.x = x$*, então* A *é denominado um* ***suporte*** *de* x*;*

(ii) Se A *é suporte de* x*, e além disso ele é um conjunto finito, então ele recebe o nome de* ***suporte finito*** *de* x*;*

(iii) X *é um* ***conjunto nominal*** *se todo elemento* x *possui um suporte finito;*

(iv) *Se x possui suporte finito, então o* ***menor suporte finito*** *de x é denotado por $supp_X(x)$, onde X pode ser omitido se ele for claro no contexto.*

Demonstração. No item (iv) dessa definição, consideramos a existência do um mínimo número de elementos para o conjunto suporte, e de fato necessitamos provar a existência desse mínimo. Se A é o único suporte finito de x então, por definição, ele será o menor suporte finito de x. Mas se x tem mais de um suporte finito, digamos A_1 e A_2, podemos verificar que $A_1 \cap A_2$ também é um suporte finito de x. Suponhamos a, a' átomos de mesmo *sort* não pertencentes a $A_1 \cap A_2$; então devemos provar que $(a\,a').x = x$. O caso trivial ocorre quando $a = a'$ (pois $(a\,a') = \iota$); No caso em que $a \neq a'$, consideremos um outro átomo a'' de mesmo *sort* de a e a', não pertencente ao conjunto finito $A_1 \cup A_2 \cup \{a, a'\}$, então $(a\,a') = (a\,a'') \circ (a'\,a'') \circ (a\,a'')$ é a uma composição de transposições tal que ela fixa x (dado que para cada um desses três pares de átomos, ambos elementos de cada par não estão em A_1 ou não estão em A_2), logo $(a\,a')$ também fixa x conforme se desejava demonstrar. Então segue imediatamente da propriedade de interseção de conjuntos, que se um elemento $x \in X$ possui um suporte finito, ele possui também um *menor suporte finito*. □

```
Variable A : AtomT.
Variable X : Set.
Variable P : PsetT A X.

Definition supports (F : aset A) (x :  X) :=
 forall a b, ~ In a F -> ~ In b F -> perm P [(a, b)] x = x.
```

Figura 2.16: *Conjunto Suporte.*

Dado um elemento de um conjunto nominal, na maioria do tempo o interesse não é saber qual o seu suporte, mas qual é conjunto (infinito) de átomos que não contém seu suporte. Em [23] o predicado *fresh* é definido conforme exibe a Figura 2.17.

Definição 2.4.9 (Fresh). *Se $x \in X$ e $y \in Y$ são elementos de conjuntos nominais e $supp_X(x) \cap supp_Y(y) = \emptyset$, então é dito que x é* ***fresh*** *(novo) para y, o que é denotado por $x \# y$.*

```
Definition freshP (a :  A) (x :  X) :=
 exists F : aset A,   In a F ∧ supports P F x.
```

Figura 2.17: *Predicado Fresh.*

A Definição 2.4.9 é bastante abrangente, pois dados quaisquer dois elementos de conjuntos nominais, é possível conceber se um é ou não novo para o outro. Basta verificar ser interseção de seus menores suportes é vazia. O Predicado `freshP` em Coq é mais restrito. Ele permite caracterizar somente o fato de um *átomo* ser novo ou não para um elemento de um conjunto nominal.

Enunciaremos o seguinte lema sem demonstrá-lo, para maiores detalhes recomendamos a leitura de [24].

Lema 2.4.10 (Freshness). *Sejam X, Y e Z três conjuntos nominais. Para todo $x \in X, y \in Y, f \in Y \to_{fs} Z, \pi \in Perm$ e átomos $a, a' \in \mathbb{A}$ de mesmo* sort*:*

(i) $\pi.((a\,a').x) = (\pi(a)\,\pi(a')).(\pi.x)$;

(ii) $x \# y \Rightarrow \pi.x \# \pi.y$;

(iii) $x \# f$ *e* $x \# y \Rightarrow x \# (f\,y)$.

Se X e Y são conjuntos nominais, então é possível obter uma ação de permutações de átomos sobre o conjunto $X \to Y$, de todas as funções de X para Y, de modo que se $f \in X \to Y$, então $\pi.f$ é igual o mapeamento de cada $x \in X$ para $\pi.(f(\pi^{-1}.x)) \in Y$. Essa definição é equivalente a ao fato da aplicação de funções ser respeitada por permutações de átomos:

$$\pi.(f(x)) = (\pi.f)(\pi.x) \tag{2.1}$$

Infelizmente, mesmo no caso de X e Y serem conjuntos nominais, não há garantia de que toda $f \in X \to Y$ possua um suporte finito. Por exemplo, se $X = \mathbb{N}$ e $Y = \mathbb{A}$, então dado que o conjunto $\mathbb{A}$ de átomos é contável, há funções sobrejetivas de $\mathbb{N}$ em $\mathbb{A}$; más não é difícil de ver que, nesse caso toda $f \in \mathbb{N} \to \mathbb{A}$ deverá ter uma imagem finita para que seu suporte seja também finito.

Todavia, se f tem um suporte finito A, então $\pi.f$ é suportada por $\{\pi(a) \mid a \in A\}$ (tal fato segue do Lema 2.4.10). Então o conjunto:

$$X \to_{fs} Y := \{f \in X \to Y \mid \exists A \text{ finito } \subseteq \mathbb{A}, A \text{ é suporte de } f\}$$

de *funções com suporte finito* de X para Y, é fechado sobre a ação de permutações de átomos e portanto é um conjunto nominal.

Uma outra maneira de formar conjuntos nominais é considerar conjuntos quocientes. Se $\sim$ é uma relação de equivalência em um conjunto nominal X, então tal como $\sim$, é subconjunto nominal de $X \times X$, o usual conjunto $X/_{\sim}$ de classes de equivalência possui uma ação de permutação átomos, dada por $\pi.[x] = [\pi.x]$. Mais adiante, uma classe de equivalência possuirá o suporte igual a qualquer representante da classe. Então $X/_{\sim}$ será um conjunto nominal. O conjunto nominal $\mathrm{T}_\alpha(\Sigma)_\sigma$ de α-termos (de aridade σ sobre uma assinatura nominal Σ) do conjunto nominal $\mathrm{T}(\Sigma)_\sigma$, é um exemplo dessa construção.

Essas classes de α-equivalência $[t]_\alpha$ de termos $t : \sigma$, têm propriedades elementares, devido a própria definição relação $=_\alpha$ (2.4.5), e assim é possível introduzir uma notação mais próxima da notação informal para as classes de α-equivalência, conforme segue:

(i) *Átomos* : Se $\mathtt{a} \in \Sigma_{\mathtt{A}}$ e $e \in \mathrm{T}_\alpha(\Sigma)_{\mathtt{a}}$, então há um único $a \in \mathbb{A}_{\mathtt{a}}$ tal que $e = [a]_\alpha$. Nesse caso, escreve-se somente a;

(ii) *α-termos construídos*: Se $\mathtt{s} \in \Sigma_{\mathtt{D}}$ e $e \in \mathrm{T}_\alpha(\Sigma)_{\mathtt{s}}$, então há um único $(K : \sigma \to \mathtt{s}) \in \Sigma_{\mathtt{C}}$ e $e' \in \mathrm{T}_\alpha(\Sigma)_\sigma$, tal que existe t' com $e' = [t']_\alpha$ e $e = [Kt']_\alpha$, nesse caso escreve-se somente Ke';

(iii) *Unidade*: $\mathrm{T}_\alpha(\Sigma)_{\mathtt{1}}$ contém uma única classe de equivalência, $[\langle\,\rangle]_\alpha$, a qual é denotada por $(\,)$;

(iv) *Pares*: Se $\sigma_1, \sigma_2 \in Ar(\Sigma)$ e $e \in \mathrm{T}_\alpha(\Sigma)_{\sigma_1 * \sigma_2}$, então há um único $e_i \in \mathrm{T}_\alpha(\Sigma)_{\sigma_i}$, para $i = 1, 2$, tal que existe t_i com $e_i = [t_i]_\alpha$ $(i = 1, 2)$ e $e = [\langle t_1, t_2\rangle]_\alpha$. Nesse caso, escreve-se somente (e_1, e_2);

(v) *Ligações de átomos*: Se $\mathtt{a} \in \Sigma_{\mathtt{A}}$, $\sigma \in Ar(\Sigma)$ e $e \in \mathrm{T}_\alpha(\Sigma)_{\ll a \gg \sigma}$, então para cada $a \in \mathbb{A}_{\mathtt{a}}$ com $a \# e$ há um único $e' \in \mathrm{T}_\alpha(\Sigma)_\sigma$ tal que existe t' com $e' = [t']_\alpha$ e $e = [\ll a \gg t']_\alpha$. Nesse caso escreve-se $a.e'$.

Nesse ponto, temos a descrição dos principais fundamentos da abordagem nominal, com a construção de um conjunto quociente, módulo uma relação de α-equivalência, alcançando uma definição precisa para os termos com ocorrências de variáveis ligadas. Mais além, nas Secções 3 e 4.2, forneceremos exemplos de aplicações de tais conceitos, no estabelecimento dos princípios de indução e recursão α-estrutural, para as respectivas gramáticas dos cálculos λ e λex.

Capítulo 3

O Cálculo λ

O Cálculo λ [8] é um sistema de reescrita de termos, que foi originalmente desenvolvido por Alonzo Church [9] na década de 30. Inicialmente Church trabalhava em uma teoria que envolvia funções e lógica, objetivando uma fundamentação teórica para tais conceitos, mas na totalidade o sistema desenvolvido continha inconsistências, tal como foi demonstrado por Kleene e Rosser em [31]. Assim o subsistema formado somente pela parte da teoria de funções, cuja a consistência foi demonstrada por Church [32], se tornou um modelo eficiente. O curioso é que, de acordo com Russell [33], a nomenclatura "Cálculo λ" surgiu devido a um erro tipográfico na publicação do trabalho de Church, que pretendia utilizar a notação $\hat{x}.2x+1$, para denotar uma função de x cujo corpo era $2x+1$. No entanto os recursos tipográficos da época induziram o posicionamento do símbolo $^\wedge$ não em cima da letra x, mas a sua esquerda, o que resultou em $^\wedge x.2x+1$. Depois em uma outra publicação apareceu $\lambda x.2x+1$, e assim o nome Cálculo λ acabou por se difundir como a nomenclatura oficial do sistema de Church.

$$a ::= x \,|\, (a\,a) \,|\, (\lambda x.a) \tag{3.1}$$

A definição acima que representar que, um termo a poderá ser uma variável; uma aplicação de termos; ou uma abstração de um termo por uma variável. A gramática do cálculo λ é baseada portanto em duas operações: *aplicação* $(a\,b)$; e *abstração* $(\lambda x.a)$, a qual representa a função de variável x e corpo a. Se a é um termo construído a partir da gramática do cálculo λ, então é notacionado que $a \in \Lambda$.

Abstrações sucessivas são associadas à direita, isto é, $\lambda x_1.\cdots\lambda x_k.a \equiv (\lambda x_1.\cdots(\lambda x_k.a))$ que é abreviado por $\lambda x_1 \ldots x_k.a$; enquanto que aplicações sucessivas são associadas à esquerda: $a_1\ \ a_2\ \ a_3 \ldots a_n \equiv (\ldots((a_1\ \ a_2)\ \ a_3)\ldots a_n)$. Por exemplo, $((y\ \ u)\ \ x)\ \ ((x\ \ u)\ \ (w\ \ v))$ é escrito como $(yux\ \ (xu\ \ (w\ \ v)))$.

```
Variable tmvar :  Set

Inductive tm :  Set :=
| var :  tmvar -> tm
| app :  tm -> tm -> tm
| lam :  tmvar -> tm -> tm.
```

Uma definição da gramática do cálculo λ em Coq, é dada acima. Mas esse tipo de definição tem um problema bem conhecido. Os construtores de uma declaração `Inductive` são sempre injetivos por definição, no entanto no caso do construtor abstração (`lam`) isso não deveria ser verdade, se quisermos incluir a relação de α-equivalência na relação de igualdade ("=") pré-construída no Coq, pois: $\lambda x.a = \lambda y.b \nrightarrow x = y \wedge a = b$. A solução dada por [23], é justamente a especificação do cálculo λ via uma abordagem nominal, a qual nós apresentamos na Seção 3.1. As funções recursivas de *variáveis livres* e *comprimento de um termo* podem ser obtidas através dos princípios de recursão ordinária (que não necessitam considerar a relação de α-equivalência entre termos).

O *abstrator* λx do termo $(\lambda x.a)$ liga todas as ocorrências da variável x em a, e qualquer outra variável que não está ligada por nenhum outro abstrator em a, é denominada variável *"livre"* do termo $(\lambda x.a)$. Então, por exemplo y ocorre livre em $\lambda x.(y\,x)$, e x ocorre ligado no termo.

Definição 3.0.11 (Conjunto de variáveis livres de um termo $a \in \Lambda$)**.** *Definimos o* ***conjunto das variáveis livres*** *de $a \in \Lambda$, denotado por* $\text{fv}(a)$*, recursivamente por:*

(i) $\text{fv}(x) := \{x\}$

(ii) $\text{fv}(a\,b) := \text{fv}(a) \cup \text{fv}(b)$

(iii) $\text{fv}(\lambda x.a) := \text{fv}(a) \setminus \{x\}$

Definição 3.0.12 (Comprimento de um termo $a \in \Lambda$)**.** *A função comprimento L de um termo $a \in \Lambda$, é definida recursivamente por:*

(i) Se $a \equiv x$, então $L(a) = 0$;

(ii) Se $a \equiv (a_0\,a_1)$, então $L(a) = L(a_0) + L(a_1) + 1$;

(iii) Se $a \equiv \lambda x.a_0$, então $L(a) = L(a_0) + 1$.

O sistema de reescrita do cálculo λ é composto de uma única regra de reescrita, que chamada β e é dada por:

Definição 3.0.13 (Regra β)**.** $(\lambda x.a)b \to_\beta a\{x/b\}$

$a\{x/b\}$ representa o termo resultante da substituição simultânea de todas as ocorrências livres de x em a por b. Essa operação é definida separadamente, através de uma função recursiva. Dada a definição da gramática do cálculo (Seção 3.1) e de tal função recursiva (Seção 3.2), é possível definir a regra β em Coq, conforme a Figura 3.1. Essa

declaração indutiva afirma que `rule_beta` é uma relação binária entre termos, pois tem tipo `tm -> tm -> Prop`, e além disso ela relaciona um a aplicação de uma abstração por um termo (`(& x, s) ! t)`) a uma meta-substituição (`s^{x := t}`). Estamos utilizando aqui uma notação criada nos arquivos da especificação, para denotar aplicações ($(a\,b)$:= `(a ! b)`), abstrações ($\lambda x.a$:= `& x, a`) e a meta-substituição ($a\{x/b\}$:= `{x := b}`) e que acreditamos facilitar o entendimento das declarações. Para a própria regra β criamos a notação: `a ->_Beta b := rule_beta a b`.

```
Inductive rule_beta : tm -> tm -> Prop :=
reg_rule_beta : forall (s t : tm)(x : tmvar),

rule_beta ((& x, s) ! t) (s^{x := t}).
```

Figura 3.1: *Definição da regra β.*

Definição 3.0.14 (Meta-substituição para λ-termos)**.** *Sendo $a, b \in \Lambda$ e x, y variáveis distintas, a* ***meta-substituição*** *$a\{x/b\}$ é definida recursivamente por:*

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

(ii) $y\{x/b\} := y$, *se* $x \neq y$

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

(iv) $(\lambda y.t)\{x/b\} = \lambda y.t\{x/b\}$, *se* $x \neq y$ *e* $y \notin \mathrm{fv}(b)$

A restrição "$x \neq y$ e $y \notin \mathrm{fv}(b)$" no item (iv) busca evitar um problema chamado *colisão entre variáveis livres e ligadas* de um termo. Consideremos o seguinte exemplo:

Exemplo 3.0.15. *O termo $\lambda y.x$ representa uma função constante onde o corpo x não tem nenhuma relação com o parâmetro y; aplicando a substituição $\{x/y\}$ a este termo de forma "ingênua" (sem observar as restrições) obtém-se $\lambda y.y$, que é a representação da função identidade, e portanto o termo obtido tem semântica distinta de $\lambda y.x$, o que seria uma inconsistência do sistema.*

3.1 Classes de α-equivalência ($\Lambda/_{=_\alpha}$)

De fato, a função da definição 3.0.14 não está definida para todo $t \in \Lambda$, pois as condições do item (iv) restringem também a propagação da recursão dentro do termo $(\lambda y.t)$. A solução dada, é primeiramente definir uma relação α-equivalência, onde qualquer termo $(\lambda z.t')$ é α-equivalente a $(\lambda y.t)$ (fato denotado por $\lambda z.t' =_\alpha \lambda y.t$) desde que z já não ocorra livre em t e t' seja igual a termo t, com o renomeamento de todas as ocorrências livres de y por z. Então por exemplo: $\lambda x.x(z\,x) =_\alpha \lambda y.y(z\,y)$. Barendregt [8] abordou essa questão, propondo uma condição de *higiene* para os termos: *"Os nomes utilizados para variáveis livres deverão ser sempre distintos dos nomes dados para variáveis ligadas em um termo".*

No entanto essa condição não dispensa o uso da relação de α-equivalência. Consideram-se então funções recursivas não sobre Λ, mas nas classes de equivalência formadas pela relação α, isto é, o conjunto $\Lambda/_{=_\alpha}$.

Deve-se lembrar que como $(\lambda x.a)$ representa uma função, cuja variável é x, logo seu renomeamento não deve gerar nenhum prejuízo de significado.

Exemplo 3.1.1. $\begin{cases} f : \mathbb{R} \to \mathbb{R} \\ f(x) = x \end{cases} \iff \begin{cases} f : \mathbb{R} \to \mathbb{R} \\ f(y) = y \end{cases}$

A abordagem nominal da seção 2.4, pode ser aplicada ao cálculo λ, comforme exibido no Eexemplo 2.4.4, permitindo a formalização dos princípios de indução e recursão sobre o conjunto quociente $\Lambda/_{=_\alpha}$. O construtor D dessa assinatura representa uma constante, a qual será chamada *dot* ($\bullet$). Já V, A e L são os respectivos construtores de *variável*, *aplicação* e *abstração*. Nesse contexto, a relação de α-equivalência entre os termos é dada por:

$$(=_\alpha .D)\,\frac{}{D =_\alpha D} \qquad (=_\alpha .V)\,\frac{a \in \mathbb{A}_{\mathtt{V}}}{V\,a =_\alpha V\,a} \qquad (=_\alpha .A)\,\frac{u =_\alpha u' : \mathtt{t} \quad \& \quad v =_\alpha v' : \mathtt{t}}{A\langle u, v\rangle =_\alpha A\langle u', v'\rangle : \mathtt{t}}$$

$$(=_\alpha .L)\,\frac{a, a', a'' \in \mathbb{A}_{\mathtt{V}} \qquad a'' \notin \{a\} \cup \mathrm{fv}(t) \cup \{a'\} \cup \mathrm{fv}(t') \qquad (a\,a'')u =_\alpha (a'\,a'')u' : \mathtt{t}}{L \ll a \gg u =_\alpha L \ll a' \gg u' : \mathtt{t}}$$

Figura 3.2: *α-equivalência no cálculo λ.*

Em [23] a definição da gramática do cálculo λ inicia por assumir que as variáveis ligadas de um termo deverão ser designadas por índices e as variáveis livres por nomes. Esse tipo de formalização é conhecida como *locally nameless representation* [34, 35], e originalmente surge em implementações para o assistente de prova *Isabelle/HOL* [4]. Nomes são definidos pela estrutura atômica conforme mostrado na Figura 2.10.

A definição indutiva `Phi` cria uma classe de termos através da qual será derivado um subconjunto de termos *bem formados*. De um modo geral, dado um termo t, seu tipo é `Phi n` se o maior índice de uma variável *pendurada* que ocorre nele, é menor que `n`. Entende-se por variável pendurada de índice `n`, uma variável do tipo `pbound n` que ocorre em um posição livre no termo. Não é difícil de se verificar que $\forall(\mathtt{n} : \mathtt{nat}), \mathtt{Phi}\,\mathtt{n} \subset \mathtt{Phi}\,(\mathtt{n}+1)$.

Assim, defini-se o conjunto de termos bem formados (`tm`) como sendo `Phi 0`, o que é exibido na Figura 3.5. Os construtores do cálculo são então identificados pelas subestruturas de `Phi` que tem tipo `Phi 0`.

Exemplo 3.1.2. *Alguns exemplos de termos definidos em* `Phi`*, são dados por:*

(i) *dot* $\in \mathtt{Phi}\,0$*;*

(ii) $x \in \mathtt{Phi}\,0$*;*

(iii) $(\lambda\,0) \in \mathtt{Phi}\,0$*;*

(iv) $0 \in$ `Phi` 1

(v) $((\lambda\, 0)(1\, x)) \in$ `Phi` 2;

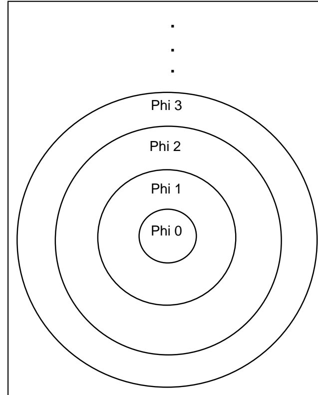


Figura 3.3: *Inclusão progressiva dos conjuntos* `Phi n`.

No caso da ocorrência de de variáveis ligadas, é necessário que se defina uma função recursiva sobre a estrutura de `Phi`, a qual substitui a ocorrência de uma variável livre em um termo `t` de tipo `Phi n`, pelo índice `n`, resultando em um termo de tipo `(n + 1)`. A função que realiza essa tarefa é chamada `abs`. E ela aparece no construtor `lam` realizando tarefa de realizar a ligação de um dado nome de variável a um termo.

```
Inductive Phi : nat -> Set :=
 | pdot : forall (n : nat), Phi n
 | pfree : forall (n : nat), tmvar -> Phi n
 | pbound : forall (n i : nat), i < n -> Phi n
 | papp : forall (n : nat), Phi n -> Phi n -> Phi n
 | plam : forall (n : nat), Phi (S n) -> Phi n.
```

Figura 3.4: *Conjunto* `Phi`.

```
Definition tm : Set := Phi 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 (x : tmvar) (b : tm) : tm := plam 0 (abs x b).
```

Figura 3.5: *Termos bem formados e construtores do cálculo λ.*

Os princípios de indução e recursão ordinária estrutural sobre termos, isto é, não considerando as classes α-equivalência, podem ser obtidos em alguns casos, somente fazendo uso estrutura indutiva de `Phi`, automaticamente gerada pelo Coq. Esse é o caso, por exemplo, das construções de funções recursivas para conjunto de variáveis livres e para o

comprimento de um termo. O princípio indução ordinária de `Phi`, e a definição da função `abs`, permitem que se demonstre o teorema da α-equivalência entre termos.

A declaração da Figura 3.6 exibe um teorema de α-equivalência, para o único construtor que tem variável ligada no cálculo λ, a abstração. Em notação matemática o enunciado do teorema se resume a: $\forall ab \in \mathbb{A}_{\mathtt{V}}, \forall s : \mathtt{t},\ b \notin \mathrm{fv}(s) \Rightarrow \lambda a.s =_\alpha \lambda b.(a\,b)s$.

```
Theorem eq_lam :
 forall a b s,
 ~ In b (fvar s) ->
 & a, s = & b, (perm tmP [(a, b)] s).
```

Figura 3.6: *Teorema de α-equivalência para o construtor abstração.*

3.2 Recursão e indução α-estrutural

Seja Σ a assinatura nominal da Figura 2.11. Então $\mathrm{T}(\Sigma)_{\mathtt{t}}$ é o conjunto de λ-termos Λ e $\mathrm{T}_\alpha(\Sigma)_{\mathtt{t}}$ é o conjunto quociente $\Lambda/_{=_\alpha}$. Podemos então utilizar a notação para estruturas com uma relação de α-equivalência definida, exibida ao final da Seção 2.4. Apresentamos então, no contexto do cálculo λ, os teoremas de indução e recursão α-estrutural, ao lado das declarações correspondentes ao código da especificação em Coq.

Teorema 3.2.1 (Teorema de Indução α-estrutural para o cálculo λ)**.** *Dado um subconjunto $S \subseteq \mathrm{T}_\alpha(\Sigma)_{\mathtt{t}}$, para provar que $S = \mathrm{T}_\alpha(\Sigma)_{\mathtt{t}}$, é suficiente mostrar que:*

(i) $D \in S$;

(ii) $\forall a \in \mathbb{A}_{\mathtt{V}},\ V\,a \in S$;

(iii) $\forall e_1\,e_2 \in S,\ A\,(e_1, e_2) \in S$;

(iv) $\exists F$ *finito* $\subseteq \mathbb{A}_{\mathtt{V}}, \forall a\,e,\ a \notin F \Rightarrow L\,a.e \in S$.

```
Theorem tm_induction :
forall (P : tm -> Type) (F : aset tmvar),
(P dot) ->
(forall a, P (var a)) ->
(forall s, P s -> forall t, P t -> P (s !  t)) ->
(forall a, ~ In a F -> forall s, P s -> P (& a, s)) ->
forall x :  tm, P x.
```

Figura 3.7: *Teorema de indução α-estrutural para o Cálculo λ.*

No Teorema 3.2.2 as condições são: primeiro o fornecimento de funções particulares, isto é, indexadas pelos construtores, todas das quais com um suporte finito F; e além disso uma condição finita para ligações (FCB), a qual exige que sempre exista um $a \in \mathbb{A}_{\mathtt{V}}$,

tal que $a \notin F$ e $\forall x,\ a \# f_L(a, x)$. A afirmação realizada é de que, existe uma única família de funções (módulo equivalência extencional), tal que essa família tem comportamento de uma função recursiva sobre $\Lambda/_{=_\alpha}$.

Teorema 3.2.2 (Teorema de Recursão α-estrutural para o cálculo λ)**.** *Suponhamos que são dados um conjunto nominal X e funções:*

$$
\begin{array}{l}
f_D \in X \\
f_V \in \mathbb{A}_{\mathtt{V}} \rightarrow_{fs} X \\
f_A \in X \times X \rightarrow_{fs} X \\
f_L \in \mathbb{A}_{\mathtt{V}} \times X \rightarrow_{fs} X
\end{array}
$$

```
Variables (R : Set) (RP : PsetT tmvar R).
Variable (f_dot :  R).
Variable (f_var :  tmvar -> R).
Variable (f_app :  tm -> R -> tm -> R -> R).
Variable (f_lam :  tmvar -> tm -> R -> R).
```

Figura 3.8: *Funções indexadas pelos construturas do Cálculo λ.*

Todas as quais com um suporte finito de átomos F. E além disso a seguinte condição é satisfeita:

$$\exists a \in \mathbb{A}_{\mathtt{V}},\ a \notin F\ \ \&\ \ \forall x \in X,\ a \# f_L(a, x).$$

```
Variable (fcb_lam :  exists b,
  ~ In b F ∧ forall x y, freshP RP b (f_lam b x y)).
```

Figura 3.9: *Condição FCB para o construtor abstração.*

Então existe uma única família de funções, de suporte finito F ($\hat{f} \in \Lambda/_{=_\alpha} \rightarrow_{fs} X$) satisfazendo:

(i) $\hat{f}(D) = f_D$;

(ii) $\hat{f}(V\,a) = f_V\,a$;

(iii) $\hat{f}(A\,(e_1,\,e_2)) = f_A\,(\hat{f}e_1,\,\hat{f}e_2)$;

(iv) $a \notin F \Rightarrow \hat{f}(L\,a.e) = f_L\,a.(\hat{f}e)$.

```
Theorem tm_rec_dot :  tm_rec dot = f_dot.
Theorem tm_rec_var :  forall a, tm_rec (var a) = f_var a.
Theorem tm_rec_app :  forall s t,
  tm_rec (s !  t) = f_app s (tm_rec s) t (tm_rec t).
Theorem tm_rec_lam :  forall a s,
  ~ In a F -> tm_rec (& a, s) = f_lam a s (tm_rec s).
```

Concluímos aqui, uma descrição da especificação de [23], do cálculo λ, via uma assinatura nominal. O passo seguinte será a construção de uma função recursiva, a meta-substituição, utilizando o Teorema 3.2.2. Esse trabalho foi desenvolvido em Coq, também por Aydemir *et al* em [23].

3.3 A operação de meta-substituição definida sobre $\Lambda/_{=_\alpha}$

A aplicação do Teorema 3.2.2 na construção da operação de meta-substituição, tem como resultado a definição exibida logo abaixo. Essa declaração fornece as funções indexadas para cada um dos construtores do cálculo, sendo que a última linha da definição afirma qual será o conjunto $F :=$ `add y (fvar s)`, suporte de todas as funções. Os enunciados dos lemas que retratam esse fato são dados logo abaixo da definição.

```
Section Meta_substitution.
Variable y : tmvar.
Variable s : tm.

Definition meta_subst : tm -> tm :=
  tm_rec tm
   dot
   (fun x => if atom_eqdec tmvar x y then s else (var x))
   (fun s s' t t' => s' ! t')
   (fun x t t' => & x, t')
   (add y (fvar s)).

Lemma meta_subst_supp_dot :
supports tmP (add y (fvar s)) dot.

Lemma meta_subst_supp_var :
supports (tmvarP ^> tmP) (add y (fvar s))
(fun x : tmvar => if atom_eqdec tmvar x y then s else var x).

Lemma meta_subst_supp_app :
supports (tmP ^> tmP ^> tmP ^> tmP ^> tmP)
(add y (fvar s)) (fun _ s' _ t' : tm => s' ! t').

Lemma meta_subst_supp_lam :
supports (tmvarP ^> tmP ^> tmP ^> tmP)
(add y (fvar s))
(fun (x : tmvar) (_ t' : tm) => & x, t').
```

É possível então concluir que a função `meta_subst` tem de fato, o comportamento desejado, ou seja, verifica-se a sequência de afirmações abaixo, quem descrevem que para

cada construtor haverá a propagação da função de forma recursiva, sendo que no caso do construtor abstração a variável ligada não poderá pertencer a conjunto F. E por fim o conjunto F é demonstrado ser também o suporte de `meta_subst`.

```
Theorem meta_subst_dot :  meta_subst dot = dot.

Theorem meta_subst_var_eq :  meta_subst (var y) = s.

Theorem meta_subst_var_neq :  forall x, x <> y ->
meta_subst (var x) = (var x).

Theorem meta_subst_app :
forall q r, meta_subst (q !  r) = (meta_subst q) !  (meta_subst r).

Theorem meta_subst_lam :
forall x t, x <> y -> ~ In x (fvar s) ->
meta_subst (& x, t) = & x, (meta_subst t).

Theorem supports_meta_subst :
supports (tmP ^-> tmP) (add y (fvar s)) meta_subst.
```

3.4 O conjunto $\mathcal{SN}_\beta$

A obtenção de um conjunto de regras indutivas para definição do conjunto $\mathcal{SN}_\beta$, é algo que possibilita a demonstração de propriedades de normalização para o próprio cálculo λ. A demonstração da PSN para o cálculo λex, utiliza também uma caracterização indutiva para o conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$.

Iniciamos com alguns exemplos de reduções no cálculo λ, que são dadas à seguir.

Exemplo 3.4.1.
$$\begin{array}{lcll} (\lambda y.\underline{(\lambda x.x\,y)\,a})\,b & \to_\beta & (\lambda y.(x\,y)\{x/a\})\,b & := & (\lambda y.(a\,y))\,b \\ \underline{(\lambda y.(a\,y))\,b} & \to_\beta & (a\,y)\{y/b\} & := & (a\,b) \end{array}$$

Exemplo 3.4.2. $(\lambda x.(x \;\; x)) \;\; (\lambda x.(x \;\; x)) \to_\beta (\lambda x.(x \;\; x)) \;\; (\lambda x.(x \;\; x))$.

Esse último exemplo mostra que o cálculo λ não é terminante, dado que $(\lambda x.(x \;\; x)) \;\; (\lambda x.(x \;\; x))$ reduz para si mesmo. Tal processo pode ser repetido indefinidamente, gerando assim um caminho de redução de comprimento ∞.

O processo de redução de um termo não é único, no Exemplo 3.4.1 havia duas possibilidades para aplicação da regra β, na raiz do termo ou no seu interior. Essas escolhas podem seguir uma lógica, ou podem vir de maneira aleatória. Quando é adotado um procedimento de escolha para qual parte o termo deverá ser primeiramente reduzida, esse processo é chamado de *Estratégia de Redução* [18].

Uma estratégia de redução do cálculo λ pode ser então classificada como:

(i) *maximal*, se ela computa para um termo, o maior caminho de redução até uma *forma normal*, caso ela exista; senão ela retorna um caminho de redução de comprimento ∞;

(ii) *minimal*, se ela computa para um termo, o menor caminho de redução até uma forma normal, caso ela exista; senão ela retorna um caminho de redução de comprimento ∞;

(iii) *perpétua*, se ela computa para um termo, um caminho de redução de comprimento ∞, caso ele exista; senão ela retorna algum caminho de redução de comprimento $n \in \mathbb{N}$ até uma forma normal, e;

(iv) *normalizante*, se ela computa para um termo, um caminho de redução comprimento $n \in \mathbb{N}$ até uma forma normal, se ela existe; senão ela retorna um caminho de redução de comprimento ∞.

Os seguintes lemas são úteis para se demonstração de que uma dada estratégia de redução é perpétua no cálculo λ, permite encontrar uma definição indutiva para o conjunto $\mathcal{SN}_\beta$.

Lema 3.4.3 (Lema Fundamental da Perpetualidade)**.** *Assumindo que* $a_1 \in \mathcal{SN}_\beta$ *sempre que* $x \notin \mathrm{fv}(a_0)$*, então para todo* $n \geq 1$[1] *:*

$$a_0\{x/a_1\}\, a_2...a_n \in \mathcal{SN}_\beta \Rightarrow (\lambda x.a_0)\, a_1...a_n \in \mathcal{SN}_\beta$$

Demonstração. Seja $a_0\{x/a_1\}\, a_2...a_n \in \mathcal{SN}_\beta$. Então $a_0, a_2, ..., a_n \in \mathcal{SN}_\beta$. Se $x \notin \mathrm{fv}(a_0)$, então, por hipótese, $a_1 \in \mathcal{SN}_\beta$. Se $x \in \mathrm{fv}(a_0)$, então a_1 ocorre em alguma posição do termo $a_0\{x/a_1\}\, a_2...a_n$, e portanto $a_1 \in \mathcal{SN}_\beta$. Se $(\lambda x.a_0)\, a_1...a_n \notin \mathcal{SN}_\beta$, então qualquer redução pertencente a um caminho infinito, deve ser da forma:

$$\begin{array}{rcl} (\lambda x.a_0)\, a_2...a_n & \overset{*}{\rightarrow}_\beta & (\lambda x.a_0')\, a_2'...a_n' \\ & \rightarrow_\beta & a_0'\{x/a_1'\}\, a_2'...a_n' \\ & \rightarrow_\beta & ... \end{array}$$

E dado que, $\forall\, a\, a'\, b\, b'$:

$$a \overset{*}{\rightarrow}_\beta a' \;\&\; b \overset{*}{\rightarrow}_\beta b' \Rightarrow a\{x/b\} \overset{*}{\rightarrow}_\beta a'\{x/b'\}$$

Então há um caminho infinito de redução dado por :

$$\begin{array}{rcl} a_0\{x/a_1\}\, a_2...a_n & \overset{*}{\rightarrow}_\beta & a_0'\{x/a_1'\}\, a_2'...a_n' \\ & \rightarrow_\beta & ... \end{array}$$

Contradizendo o fato de que $a_0\{x/a_1\}\, a_2...a_n \in \mathcal{SN}_\beta$.

□

[1] $(n = 1) \Rightarrow (a_0\{x/a_1\}\, a_2...a_n \equiv a_0\{x/a_1\})$

Corolário 3.4.4. *Se $a_1 \in \mathcal{SN}_\beta$, então para todo $n \geq 1$:*

$$a_0\{x/a_1\}\, a_2...a_n \in \mathcal{SN}_\beta \Rightarrow (\lambda x.a_0)\, a_1...a_n \in \mathcal{SN}_\beta$$

Demonstração. Aplicação direta do Lema 3.4.3.

□

Lema 3.4.5 (Caracterização Indutiva de $\mathcal{SN}_\beta$)**.** *Seja X definido indutivamente por:*

(i) $a_1, ..., a_n \in X \Rightarrow x\, a_1...a_n \in X$*;*
(ii) $a \in X \Rightarrow \lambda x.a \in X$*;*
(iii) $a_1 \in X$ *e* $a_0\{x/a_1\}\, a_2...a_n \in X \Rightarrow (\lambda x.a_0)\, a_2...a_n \in X$*;*

Então $\mathcal{SN}_\beta = X$.

Demonstração. É necessário demonstrar a dupla inclusão $\mathcal{SN}_\beta \subseteq X$ e $X \subseteq \mathcal{SN}_\beta$.
($a \in \mathcal{SN}_\beta \Rightarrow a \in X$) pode ser demonstrada não somente pelas possíveis derivações de a em $\mathcal{SN}_\beta$, mas através da indução lexicográfica no par $\langle l_\beta(a), L(a)\rangle$:

1. $a \equiv x\, b_1...b_n$. Então $b_1, ..., b_n \in \mathcal{SN}_\beta$. E por hipótese de indução $b_1, ..., b_n \in X$, logo $a \in X$.
2. $a \equiv \lambda x.b$. Demonstração é análoga ao caso anterior.
3. $a \equiv (\lambda x.b_0)\, b_1...b_n$. Então $b_1 \in \mathcal{SN}_\beta$, $b_0\{x/b_1\}\, b_2...b_n \in \mathcal{SN}_\beta$, logo por hipótese de indução, $b_1 \in X$ e $b_0\{x/b_1\}\, b_2...b_n \in X$, então $a \in X$.

Para ($a \in X \Rightarrow a \in \mathcal{SN}_\beta$) a demostração é realizada de indução na estrutura de X:

1. $a \equiv x\, b_1...b_n$. Então $b_1, ..., b_n \in X$. E por hipótese de indução $b_1, ..., b_n \in \mathcal{SN}_\beta$, logo $a \in \mathcal{SN}_\beta$.
2. $a \equiv \lambda x.b$. Demonstração é análoga ao caso anterior.
3. $a \equiv (\lambda x.b_0)\, b_1...b_n$. Então $b_1 \in X$, $b_0\{x/b_1\}\, b_2...b_n \in X$, logo por hipótese de indução, $b_1 \in \mathcal{SN}_\beta$ e $b_0\{x/b_1\}\, b_2...b_n \in \mathcal{SN}_\beta$, então $a \in \mathcal{SN}_\beta$.

□

Capítulo 4

Cálculos de substituições explícitas

No cálculo λ, 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 cálculo λ é uma operação em um meta-nível, descrita fora da gramática do cálculo, em cálculos de substituições explícitas este processo é internalizado e manipulado por símbolos e regras de redução, dentro da própria sintaxe do sistema, objetivando fornecer um formalismo adequado e mais próximo das implementações reais.

Cálculos de substituições explícitas, tem aplicações em diferentes áreas da computação, tal como programação lógica e funcional, teoria da prova, formalização de teoremas, linguagens orientadas a objeto, dentre outras. Existem diversas formas de se definir um cálculo de substituições explícitas, todavia algumas restrições são bastante relevantes. Por exemplo, a definição da regra β em 3.0.13 e da substituição em 3.0.14, são limites bem claros nesse contexto.

Há de se verificar que os novos formalismos, que implementam substituições explícitas, atendem simultaneamente a determinadas propriedades, como a *simulação de um passo da β-redução*, a *confluência em termos abertos e fechados*[1], a *preservação da normalização forte* e a *composição completa*, o que é algo não muito comum.

4.1 Cálculos em notação de *de Bruijn* ($\lambda\sigma$, $\lambda\sigma_{\Uparrow}$)

Um primeiro exemplo de cálculo de substituições explícitas é o $\lambda\sigma$ [1], o qual utiliza índices de *de Bruijn* [36] para designação de variáveis. Essa notação considera qualquer variável como um índice natural $\underline{\mathtt{n}}$.

A gramática do $\lambda\sigma$ é um pouco mais complexa do que a do cálculo λ, pois ela tem duas classes de objetos : *termos* e *substituições*. A notação com índices de *de Bruijin* tem vantagens, por exemplo ela dispensa o uso de α-conversão, já que um termo do tipo $\lambda\underline{1}$ representa uma classe de equivalência, de todos os termos $\lambda x.x$, onde x pode ser renomeado

[1] Termos fechados são aqueles que não possuem *metavariáveis*, e abertos são os que possuem. Metavariável é um conceito que é definido para cada cálculo de modo particular, mas sempre representa um "buraco", isso é, um lugar no termo que pode ser preenchido por uma variedade de outros termos.

por qual outra variável. Em contrapartida os cálculos com índices adicionam uma dificuldade com relação a manipulação de sua notação, que é mais adequada à máquinas do que a humanos.

$$\begin{array}{lcll} \text{TERMOS} & M,\ N & := & \underline{1} \mid X \mid \lambda M \mid (MN) \mid M[S] \\ \text{SUBSTITUIÇÕES} & S,\ T & := & id \mid \uparrow \mid M.S \mid S \circ T \end{array} \tag{4.1}$$

(Beta)	$(\lambda M\, N)$	$\to$	$M[N.id]$
(App)	$(M\ N)[S]$	$\to$	$(M[S]\, N[S])$
(Abs)	$(\lambda M)[S]$	$\to$	$\lambda M[\underline{1}.(S \circ \uparrow)]$
(Clos)	$M[S][T]$	$\to$	$M[S \circ T]$
(Varcons)	$\underline{1}\ [M.S]$	$\to$	M
(Id)	$M[id]$	$\to$	M
(Assoc)	$(S_1 \circ S_2) \circ T$	$\to$	$S_1 \circ (S_2 \circ T)$
(Map)	$(M.S) \circ T$	$\to$	$M[T](S \circ T)$
(IdL)	$id \circ S$	$\to$	S
(IdR)	$S \circ id$	$\to$	S
(ShiftCons)	$\uparrow \circ (M.S)$	$\to$	S
(VarShift)	$\underline{1}. \uparrow$	$\to$	id
(Scons)	$\underline{1}\ [S].(\uparrow \circ S)$	$\to$	S
($\mathtt{Eta}_\sigma$)	$\lambda(M\ \underline{1})$	$\to$	N, se $M =_\sigma N[\uparrow]$

Figura 4.1: *Regras de reescrita do Cálculo $\lambda\sigma$.*

Exemplo 4.1.1. *O construtor $\uparrow$ é definido de forma que, para um natural $n \in \mathbb{N}$, temos $\underline{\mathtt{n}}[\uparrow]^0 := \underline{\mathtt{n}}$ e $\underline{\mathtt{n+1}} := \underline{1}[\uparrow]^n$. Assim, por exemplo o termo $\lambda z.\lambda x.(x\, x)\, z$ é representado por $\lambda\lambda(\underline{1}\, \underline{1})\, \underline{1}[\uparrow]$ e $\lambda w.\lambda z.\lambda x.(x\, z)\, w$ é representado por $\lambda\lambda\lambda(\underline{1}\, \underline{1}[\uparrow])\, \underline{1}[\uparrow]^2$.*

O conjunto de regras do $\lambda\sigma$ pretende simular a regra β, sendo que o precesso é estartado pela regra (Beta), do $\lambda\sigma$, e a operação de substituição é executada pela aplicação das demais regras.

Exemplo 4.1.2. *Segue um exemplo de simulação da regra β no cálculo $\lambda\sigma$:*

$(\lambda x.(x\, x)\, z$	$\to_\beta$	$(x\, x)\{x/z\} := (z\, z)$
$(\lambda(\underline{1}\, \underline{1}))\, \underline{1}[\uparrow]^2$	$\to_{(\mathtt{Beta})}$	$(\underline{1}\, \underline{1})[\underline{1}[\uparrow]^2.id]$
$(\underline{1}\, \underline{1})[\underline{1}[\uparrow]^2.id]$	$\to_{(\mathtt{App})}$	$((\underline{1}[\underline{1}[\uparrow]^2.id])\, (\underline{1}[\underline{1}[\uparrow]^2.id]))$
$((\underline{1}[\underline{1}[\uparrow]^2.id])\, (\underline{1}[\underline{1}[\uparrow]^2.id]))$	$\to_{(\mathtt{Varcons})}$	$(\underline{1}[\uparrow]^2\, \underline{1}[\uparrow]^2) \equiv (z\, z)$

No $\lambda\sigma$ a meta-substituição, isso é, aquela equivalente a que é gerada na regra β do cálculo λ, é definida para a gramática do novo cálculo através de recursão sobre a estrutura dos termos.

Definição 4.1.3 (Meta-substituição no cálculo $\lambda\sigma$)**.** *Para todo termo M do cálculo $\lambda\sigma$, a função* ***meta-substituição*** *é definida recursivamente por:*

(i) $(M_1 M_2)\{\underline{\mathtt{n}}/N\} := M1\{\underline{\mathtt{n}}/N\}\, M2\{\underline{\mathtt{n}}/N\}$

(ii) $(\lambda M)\{\underline{\mathtt{n}}/N\} := \lambda M\{\underline{\mathtt{n+1}}/N^{+}\}$

(iii) $\underline{\mathtt{m}}\{\underline{\mathtt{n}}/N\} := \begin{cases} \underline{\mathtt{m}-1} & \text{, se } \mathtt{m} > \mathtt{n} \\ N & \text{, se } \mathtt{m} = \mathtt{n} \\ \underline{\mathtt{m}} & \text{, se } \mathtt{m} \leq \mathtt{n} \end{cases}$

Onde M^{+i} também é definido recursivamente por:

(a) $(MN)^{+i} := (M^{+i}N^{+i})$

(b) $(\lambda M)^{+i} = \lambda(M^{+(i+1)})$

(c) $\underline{\mathtt{n}}^{+i} := \begin{cases} \underline{\mathtt{n}+1} & \text{, se } \mathtt{n} > i \\ \underline{\mathtt{n}} & \text{, se } \mathtt{n} \leq i \end{cases}$

Observação: *A elevação de um termo M à sua 0-potência é denotada por M^{+}.*

Exemplo 4.1.4. *Exibimos uma comparação entre as operações de subtiuição no cálculo λ e no cálculo $\lambda\sigma$:*

$$\begin{array}{llllllll} (\lambda x.y\,x)\{y/z\,w\} & = & \lambda x.(y\,x)\{y/z\,w\} & = & \lambda x.y\{y/z\,w\}\,x\{y/z\,w\} & = & \lambda x.(z\,w)\,x \\ \hline (\lambda\underline{2}\,\underline{1})\{\underline{1}/\underline{3}\,\underline{4}\} & = & \lambda(\underline{2}\,\underline{1})\{\underline{1+1}/(\underline{3}\,\underline{4})^{+}\} & = & \lambda\underline{2}\{\underline{2}/(\underline{4}\,\underline{5})\}\,\underline{1}\{\underline{2}/(\underline{4}\,\underline{5})\} & = & \lambda(\underline{4}\,\underline{5})\,\underline{1} \end{array}$$

O cálculo $\lambda\sigma$ recebeu ainda uma extensão, a qual foi denominada $\lambda\sigma_{\Uparrow}$ [37]. Nós a citamos nesse texto, não somente porque ela é um exemplo importante como comparação com os demais cálculos, mas pelo fato de que o cálculo $\lambda\sigma_{\Uparrow}$ já possui uma especificação em Coq, realizada [25], a qual serve de referência para as novas especificações de cálculos de substituições explícitas. Inicialmente nós adaptamos a especificação de [25] para definir a gramática do cálculo $\lambda\sigma$, o que foi uma tarefa relativamente simples, pois as definições indutivas do Coq a realizam de forma plena, gerando automaticamente os princípios de indução e recursão estrutural. Dado que o $\lambda\sigma$ e $\lambda\sigma_{\Uparrow}$ são cálculos de designação de variáveis por índices de *de Bruijin*, não é necessário definir em suas gramáticas, conceitos como α-equivalência e o conjunto quociente definido por essa relação. Quando avançamos, optando por trabalhar com o cálculo λex, que lida com nomes ao invés de índices, tivemos que estudar os conceitos desenvolvidos nas secções 2.4 e 3.2, os quais aparecem aplicados na especificação do cálculo λ em [23].

4.2 Cálculos com *nomes* (λx, λex, λes)

Uma maneira natural de especificar o cálculo λ, com substituições explícitas, é a codificação explícita da definição 3.0.14, tal que ela continue funcionando, módulo α-equivalência, o que resumidamente, resulta no cálculo λx [38, 39, 40, 41]:

$$a ::= x \mid (a\, a) \mid (\lambda x.a) \mid a[x/a] \tag{4.2}$$

```
Inductive Phi : nat -> Set :=
 | pdot : forall (n : nat), Phi n
 | pfree : forall (n : nat), tmvar -> Phi n
 | pbound : forall (n i : nat), i < n -> Phi n
 | papp : forall (n : nat), Phi n -> Phi n -> Phi n
 | plam : forall (n : nat), Phi (S n) -> Phi n
 | psubst : forall (n : nat), Phi (S n) -> Phi n -> Phi n.
```

Figura 4.2: *Conjunto* `Phi` *acrescido do construtor da substituição explícita.*

Se s é construído a partir da definição indutiva (4.2), então esse fato é notacionado por $s \in \mathcal{T}$. Adaptamos a especificação de [23] para esse nova gramática, incluindo nela o construtor da substituição explícita ($a[x/a]$). O primeiro passo foi adicionar um novo construtor na definição indutiva de `Phi`, da especificação. A α-equivalência é estendida para o caso do construtor da substituição explícita, onde por exemplo: $(\lambda y.x)[x/y] =_\alpha (\lambda z.x)[z/y]$. O que é representado através dos teoremas da Figura 4.3.

```
Theorem eq_lam :
 forall a b s,
 ~ In b (fvar s) ->
 & a, s = & b, (perm tmP [(a, b)] s).

Theorem eq_subst :
 forall a b s t,
 ~ In b (fvar s) ->
 s ^[a | t] = (perm tmP [(a, b)] s) ^[b | t] .
```

Figura 4.3: *Teorema de α-equivalência para a abstração e a substituição explícita.*

Da mesma forma que na especificação do cálculo λ, os princípios de recursão ordinária, inerentes a definição de `Phi`, permitem a construção das funções recursivas das variáveis livres e do comprimento de um termo.

Definição 4.2.1 (Conjunto das variáveis livres de $t \in \mathcal{T}$)**.** *O conjunto das variáveis livres de um dado termo $t \in \mathcal{T}$, notacionados por* $\mathrm{fv}(t)$*, é definido indutivamente por:*

$$
\begin{array}{lcl}
\mathrm{fv}(x) & := & \{x\} \\
\mathrm{fv}(\lambda x.u) & := & \mathrm{fv}(u) \setminus \{x\} \\
\mathrm{fv}(u\,v) & := & \mathrm{fv}(u) \cup \mathrm{fv}(V) \\
\mathrm{fv}(u[x/v]) & := & (\mathrm{fv}(u) \setminus \{x\}) \cup \mathrm{fv}(v)
\end{array}
$$

Definição 4.2.2 (Comprimento de $t \in \mathcal{T}$). *A função comprimento L de um termo $t \in \mathcal{T}$, é definida recursivamente por:*

(i) Se $t \equiv x$, então $L(t) = 0$;

(ii) Se $t \equiv (s\,v)$, então $L(t) = L(s) + L(v) + 1$;

(iii) Se $t \equiv \lambda x.s$, então $L(t) = L(s) + 1$;

(iv) Se $t \equiv s[x/v]$, então $L(t) = L(s) + L(v) + 1$.

Portanto, $\lambda x.t$ e $t[x/u]$ ligam as ocorrências livres de x em t. É usada a notação $\overline{t_n}$ para a lista de n ($n \leq 0$) termos $t_1, ..., t_n$ e $u\overline{t_n}$ para $ut_1...t_n$, o que é uma abreviação de $(...((ut_1)\,t_2)...\,t_n)$.

$$
\begin{array}{llll}
(\lambda x.t)u & \rightarrow_{\mathtt{B}} & t[x/u] & \\
x[x/u] & \rightarrow_{\mathtt{Var}} & u & \\
t[x/u] & \rightarrow_{\mathtt{Gc}} & t & se\ x \notin \mathrm{fv}(t) \\
(tu)[x/v] & \rightarrow_{\mathtt{App}} & t[x/v]\,u[x/v] & \\
(\lambda y.t)[x/v] & \rightarrow_{\mathtt{Lamb}} & \lambda y.t[x/v] &
\end{array}
$$

Figura 4.4: *Regras de reescrita do Cálculo* λx.

O sistema de reescrita acima (Figura 4.4) 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 cálculo λ é confluente, o cálculo aqui definido sofre de no mínimo um bem conhecido exemplo de divergência não juntável:

$$
t[y/v][x/u[y/v]] \overset{*}{\leftarrow} ((\lambda x.t)u)[y/v] \overset{*}{\rightarrow} t[x/u][y/v] \tag{4.3}
$$

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

$$
t[x/u][y/v] \quad \rightarrow_{\mathtt{Comp}} \quad t[y/v][x/u[y/v]] \qquad se\ y \in \mathrm{fv}(u) \tag{4.4}
$$

A restrição $y \in \mathrm{fv}(u)$ em (4.4) é feita, justamente porque nesse caso específico, a regra: $t[x/u][y/v] \rightarrow_{\mathtt{Comp}} t[y/v][x/u[y/v]]$ é fundamentalmente necessária para garantia da confluência. Além disso, tomando-se somente o sistema x (sistema formado por todas as regras do cálculo, exceto a regra B) e não se inserindo a restrição $y \in \mathrm{fv}(u)$ à aplicação da regra (4.4), pode-se ter um caso de não terminância, quando $y \notin \mathrm{fv}(u)$ e $x \notin \mathrm{fv}(v)$:

$$t[x/u][y/v] \to_{\texttt{Comp}} t[y/v][x/\underline{u[y/v]}] \to_{\texttt{Gc}} t[y/v][x/u] \to_{\texttt{Comp}} t[x/u][y/\underline{v[x/u]}] \to_{\texttt{Gc}}$$
$$t[x/u][y/v] \to_{\texttt{Comp}} \dots$$

A regra (4.4) nada mais é que uma regra de composição para substituições explícitas. Esse tipo de regra aparece primeiramente no $\lambda\sigma$, e dentre outras coisas ela é importante para estabelecer a confluência em termos abertos [10, 37]. Infelizmente o λx não é confluente em termos abertos (termos com meta-variáveis), o que é uma deficiência desse sistema.

A equação $t[x/u][y/v] =_{\texttt{C}} t[y/v][x/u]$ *se* $y \notin \text{fv}(u)$ & $x \notin \text{fv}(v)$, adiciona, ao mesmo tempo, composição completa e confluência em meta-termos ao λx, e é anexada ao sistema da Figura 4.4 juntamente com (4.4), resultando na definição do conjunto de regras de reescrita do λex. Muitos cálculos de substituições explícitas tal como o $\lambda\sigma$, $\lambda\sigma_{\Uparrow}$, tem composição completa, mas não preservam a normalização forte, como foi constatado por Mèllies [42]. Mesmo assim, a composição completa e a normalização forte podem ocorrer juntas; este é, por exemplo, o caso do cálculo λex e do λes [10].

No Coq definimos o conjunto de regras através de declarações indutivas, conforme exemplifica a Figura 4.7. A equação `C` é declarada na forma de um axioma na especificação.

```
Axiom equation_C : forall (t u v:tm) (x y:tmvar),
~ In y (fvar u) -> ~ In x (fvar v) ->
t ^[x | u] ^[y | v] = t ^[y | v] ^[x | u].
```

Figura 4.5: *Equação* `C`*.*

Equação:

$t[x/u][y/v]$	$=_{\texttt{C}}$	$t[y/v][x/u]$	*se* $y \notin \text{fv}(u)$ & $x \notin \text{fv}(v)$

Regras:

$(\lambda x.t)u$	$\to_{\texttt{B}}$	$t[x/u]$	
$x[x/u]$	$\to_{\texttt{Var}}$	u	
$t[x/u]$	$\to_{\texttt{Gc}}$	t	*se* $x \notin \text{fv}(t)$
$(tu)[x/v]$	$\to_{\texttt{App}}$	$t[x/v]\, u[x/v]$	
$(\lambda y.t)[x/v]$	$\to_{\texttt{Lamb}}$	$\lambda y.t[x/v]$	
$t[x/v][y/v]$	$\to_{\texttt{Comp}}$	$t[y/v][x/u[y/v]]$	*se* $y \in \text{fv}(u)$

Figura 4.6: *Regras de reescrita do cálculo* λex*.*

A relação de redução $\to_{\texttt{x}}$ é a relação gerada pela aplicação de qualquer uma das regras da Figura 4.6, exceto a regra `B`, em qualquer posição de um termo: na raiz, no lado esquerdo ou direito de uma aplicação, no corpo de uma abstração e no corpo ou no argumento de uma substituição explícita. Denota-se `Bx` para a relação dada por $\texttt{B} \cup \texttt{x}$. A *relação de equivalência* $=_{\texttt{e}}$ é gerada pela união $=_\alpha \cup =_{\texttt{C}}$. De forma que, as *relações de redução* (módulo $=_{\texttt{e}}$) $\to_{\texttt{ex}}$ e $\to_{\lambda\texttt{ex}}$ são respectivamente definidas por:

$$t \to_{\texttt{ex}} t' \quad \text{sse} \quad \exists s, s' \quad \text{tal que} \quad t =_{\texttt{e}} s \to_{\texttt{x}} s' =_{\texttt{e}} t'$$
$$t \to_{\lambda\texttt{ex}} t' \quad \text{sse} \quad \exists s, s' \quad \text{tal que} \quad t =_{\texttt{e}} s \to_{\texttt{Bx}} s' =_{\texttt{e}} t'$$

```
Inductive rule_b: tm -> tm -> Prop :=
reg_rule_b : forall (s t : tm)(x : tmvar),
rule_b ((& x, s) ! t) (s^[x | t]).
Notation "M ->_B N":= (rule_b M N)
(at level 59, left associativity).

Inductive rule_var: tm -> tm -> Prop :=
reg_rule_var : forall (s t : tm)(x : tmvar),
rule_var (x^[x | t]) t.
Notation "M ->_Var N":= (rule_var M N)
(at level 59, left associativity).

Inductive rule_gc : tm -> tm -> Prop :=
reg_rule_gc : forall (s t : tm) (x : tmvar),
~ In x (fvar s) -> rule_gc (s ^[x | t]) s.
Notation "M ->_Gc N":= (rule_gc M N)
(at level 59, left associativity).

Inductive rule_app : tm -> tm -> Prop :=
reg_rule_app : forall (s t u: tm) (x : tmvar),
rule_app ((s ! t) ^[x | u]) ((s ^[x | u]) ! (t ^[x | u])).
Notation "M ->_App N":= (rule_app M N)
(at level 59, left associativity).

Inductive rule_lamb : tm -> tm -> Prop :=
reg_rule_lamb : forall (s t : tm) (x y : tmvar),
x <> y -> ~ In x (fvar t) ->
rule_lamb ((& x, s) ^[y | t]) (& x, (s ^[y | t])).
Notation "M ->_Lamb N":= (rule_lamb M N)
(at level 59, left associativity).

Inductive rule_comp : tm -> tm -> Prop :=
reg_rule_comp : forall (s t u : tm) (x y : tmvar),
x <> y -> ~ In x (fvar u) -> In y (fvar t) ->
rule_comp ((s ^[x | t]) ^[y | u]) ((s ^[y | u]) ^[x | (t ^[y | u])]).
Notation "M ->_Comp N":= (rule_comp M N)
(at level 59, left associativity).
```

Figura 4.7: *Conjunto de regras do* λex *no Coq.*

No Coq, definimos primeiramente o `sistema_ex` que representa a aplicação de qualquer uma das regras do λex na raiz de um termo, excetuando-se a regra `B`.

```
Inductive sistema_ex :  tm -> tm -> Prop :=
 | rule_var_ex :  forall M N: tm, (M ->_Var N) -> sistema_ex M N
 | rule_gc_ex :  forall M N: tm, (M ->_Gc N) -> sistema_ex M N
 | rule_app_ex :  forall M N: tm, (M ->_App N) -> sistema_ex M N
 | rule_lamb_ex :  forall M N: tm, (M ->_Lamb N) -> sistema_ex M N
 | rule_comp_ex :  forall M N: tm, (M ->_Comp N) -> sistema_ex M N.
Notation "M ->_[ex] N":= (sistema_ex M N)
(at level 59, left associativity).
```

Figura 4.8: *Sistema de reescrita* `ex`.

O `sistema_Bex` é então definido para representar a aplicação de qualquer uma das regras do `sistema_ex` adicionando-se a regra `B`, a raiz de um termo.

```
Inductive sistema_Bex :  tm -> tm -> Prop :=
 | rel_rule_b_Bex :  forall M N: tm, (M ->_B N) -> sistema_Bex M N
 | rel_sistema_ex :  forall M N: tm, (M ->_[ex] N) -> sistema_Bex M N.
Notation "M ->_[Bex] N":= (sistema_Bex M N)
(at level 59, left associativity).
```

Figura 4.9: *Sistema de reescrita* `Bex`.

As relações de redução `rel_ex` e `rel_lex`, são dadas respetivamente pela aplicação dos sistemas `ex` e `Bex`, em qualquer posição de um termo. Na Figura 4.10 exibimos o início da definição indutiva da relação `rel_lex`.

```
Inductive rel_lex :  tm -> tm -> Prop :=
 | rel_root_lex :  forall (a b:tm), a ->_[Bex] b -> rel_lex a b

 | rel_app_fun_lex :  forall (a b b':tm),
   rel_lex a b -> rel_lex (a !  b') (b !  b')

 | rel_app_arg_ex :  forall (a b b':tm),
   rel_ex a b -> rel_ex (b'!  a) (b' !  b)

 | rel_lam_ex :  forall (x:tmvar) (a b:tm),
   rel_ex a b -> rel_ex (& x, a) (& x, b)

 | rel_subst_fun_ex :  forall (x:tmvar) (a b b':tm),
   rel_ex a b -> rel_ex (a ^[x | b']) (b ^[x | b'])

 | rel_subst_arg_ex :  forall (x:tmvar) (a b b':tm),
   rel_ex a b -> rel_ex (b' ^[x | a]) (b' ^[x | b]).

Notation "M ->_ex N":= (rel_ex M N)
(at level 59, left associativity).
```

Figura 4.10: *Relação de redução* λ`ex`.

Como exemplo de extensão do cálculo λex, exibimos o cálculo λes. Ele utiliza a mesma gramática do λex, mas possui um maior número de regras reescrita, buscando exercer um maior controle na propagação das substituições explícitas. Esse controle diminui o número de passos de redução de um termo, mas em contrapartida gera um maior custo computacional, dado que são realizadas repetidas chamadas da função recursiva "fv", no processo de reescrita.

Equações:			
$t[x/u][y/v]$	$=_C$	$t[y/v][x/u]$	*se* $y \notin \text{fv}(u)$ & $x \notin \text{fv}(v)$
Regras:			
$(\lambda x.t)u$	$\rightarrow_B$	$t[x/u]$	
$x[x/u]$	$\rightarrow_{Var}$	u	
$t[x/u]$	$\rightarrow_{Gc}$	t	*se* $x \notin \text{fv}(t)$
$(tu)[x/v]$	$\rightarrow_{App_1}$	$(t[x/v]\, u[x/v])$	$x \in \text{fv}(t)$ & $x \in \text{fv}(u)$
$(tu)[x/v]$	$\rightarrow_{App_2}$	$(t\, u[x/v])$	$x \notin \text{fv}(t)$ & $x \in \text{fv}(u)$
$(tu)[x/v]$	$\rightarrow_{App_3}$	$t[x/v]\, u$	$x \in \text{fv}(t)$ & $x \notin \text{fv}(u)$
$(\lambda y.t)[x/v]$	$\rightarrow_{Lamb}$	$\lambda y.t[x/v]$	
$t[x/v][y/v]$	$\rightarrow_{Comp_1}$	$t[y/v][x/u[y/v]]$	*se* $y \in \text{fv}(u)$ & $y \in \text{fv}(t)$
$t[x/v][y/v]$	$\rightarrow_{Comp_2}$	$t[x/u[y/v]]$	*se* $y \in \text{fv}(u)$ & $y \notin \text{fv}(t)$

Figura 4.11: *Sistema de regras do cálculo λes.*

Realizando agora uma rápida comparação entre os cálculos de substituições explícitas apresentados, com respeito as características desejadas para uma extensão do cálculo λ; consideramos λ_z uma extensão do cálculo λ, e a função (f_λ) da sintaxe λ para a sintaxe λ_z (algumas vezes esta função é justamente a identidade). É interessante observar se λ_z possui algumas (ou a totalidade) das 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 meta-termos é desejável, dado que muitas vezes se quer elaborar provas com *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 normalizáveis no cálculo λ, também o são no λ_z;

SIM Um passo de β-redução no cálculo λ pode ser simulado por λ_z. Ou seja, se $t \rightarrow_\beta t'$ então $f_\lambda(t) \rightarrow^*_{\lambda_z} f_\lambda(t')$;

FC A substituição explícita definida em λ_z efetivamente implementa a operação de meta-substituição do cálculo λ.

Adaptamos aqui uma tabela realizada em [10] que aponta, para alguns cálculos de substituições explícitas, a presença ou não, das propriedades acima mencionadas:

Cálculo	*C*	*MC*	*PSN*	*SIM*	*FC*
λx [38, 39, 40, 41], λ_s [43]	*Sim*	*Não*	*Sim*	*Sim*	*Não*
λex, λes [10]	*Sim*	*Sim*	*Sim*	*Sim*	*Sim*
$\lambda\sigma$ [1]	*Sim*	*Não*	*Não*	*Sim*	*Sim*
$\lambda\sigma_{\Uparrow}$ [37], λ_{se} [43]	*Sim*	*Sim*	*Não*	*Sim*	*Sim*

Figura 4.12: *Tabela comparativa.*

Concluímos que os cálculos λex e λes, apesar da simplicidade de notação de suas regras, tem as importantes propriedades desejadas para uma extensão do cálculo λ, e além disso eles cumprem a tarefa de implementar as substituições explícitas.

4.3 Uma gramática auxiliar (λex')

No trabalho de extensão da especificação de [23] para a gramática do λex, não tivemos dificuldade em demonstrar o Teorema de indução α-estrutural (Figura 4.13), pois sua adaptação possui um grau de complexidade relativamente baixo. Contudo, quando buscamos realizar o mesmo trabalho com o Teorema de recursão α-estrutural, nos deparamos com uma barreira na demonstração de um dos lemas finais da seção de recursão.

O problema aparece quando temos que permutar a variável ligada da substituição explícita, pois quando aplicamos a permutação `[(x, y)]`, com `y` $\notin$ fv(`s`)$\cup$ fv(`t`), nós temos possivelmente `s^[x | t]` $\neq_\alpha$ `([(x, y)]s)^[y | [(x, y)]t]`, dado que `x` pode ocorrer livre em `t`. Buscamos algumas alternativas na demostração do lema que estava pendente, mas não obtivemos sucesso.

```
Theorem tm_induction :
forall (P : tm -> Type) (F : aset tmvar),
(P dot) ->
(forall a, P (var a)) ->
(forall s, P s -> forall t, P t -> P (s !  t)) ->
(forall a, ~ In a F -> forall s, P s -> P (& a, s)) ->
(forall a, ~ In a F -> forall s, P s -> forall t, P t ->
  P (s ^[a | t])) ->
forall x :  tm, P x.
```

Figura 4.13: *Teorema de indução α-Estrutural para o cálculo* λex.

Nossa principal intenção é definir a função recursiva de meta-substituição para o cálculo λex, o que é muito semelhante ao que ocorre na Definição 3.0.14 para cálculo λ. Mas é necessário levar em consideração um novo caso, para a situação onde o termo é uma substituição explícita, isto é, a recursão em $t[y/u]\{x/v\}$.

Definição 4.3.1 (Meta-substituição no Cálculo λex). *Sendo $u, b \in \mathcal{T}$ e x, y variáveis distintas, a **meta-substituição** $u\{x/b\}$ 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) $(\lambda y.t)\{x/b\} = \lambda y.t\{x/b\}$, *se* $y \notin \mathrm{fv}(b)$

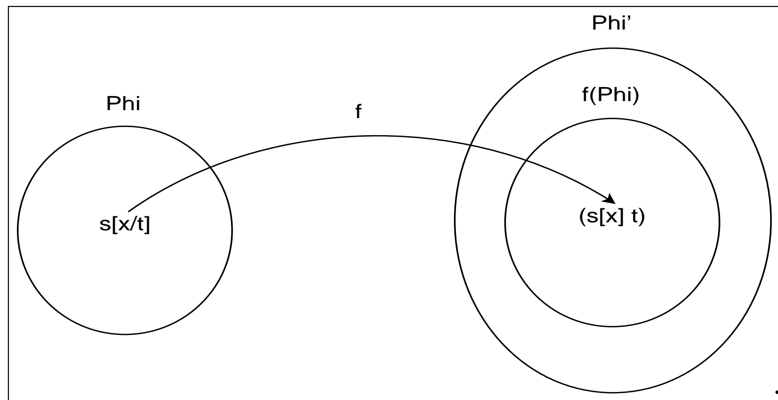
(v) $t[y/u]\{x/b\} = t\{x/b\}[y/u\{x/b\}]$, *se* $y \notin \mathrm{fv}(b)$

Observamos que processo de recursão em: $t[y/u]\{x/v\}$; é muito semelhante ao que ocorre no *β-redex*: $((\lambda y.t)\,u)\{x/v\}$. Estabelecemos uma gramática paralela, a qual chamamos de um *gramática auxiliar* (λex'), onde a substituição explícita fosse substituída por uma espécie de segunda abstração, ou seja, uma substituição que não carrega o seu argumento, por exemplo $s[x]$. Para aplicar a função de meta-substituição a um termo $u \in \mathcal{T}$, transladamos u para gramática auxiliar, onde foram realizadas as chamadas recursivas da meta-substituição definida, e por fim transladamos o resultado de volta para gramática do λex.

λex	λex'
$(x \mid (a\,a) \mid \lambda x.a \mid \mathbf{a[x/a]})$	$(x \mid (a\,a) \mid \lambda x.a \mid \mathbf{a[x]})$

Figura 4.14: λex *e* λex'.

Criamos então uma função recursiva **f** que leva cada termo da gramática do λex na gramática auxiliar, mantendo a maioria termos fixos, exceto quando $(s[x/t])$ é levada na aplicação de $s[x]$ por t, conforme ilustra o diagrama e o código abaixo.




```
Fixpoint f_Phi (n : nat) (x : Phi n) struct x : (Phi' n) :=
 match x with
  | pdot _ => pdot' _
  | pfree _ a => pfree' _ a
  | pbound _ i pf => pbound' _ i pf
  | papp _ s t => papp' _ (f_Phi _ s) (f_Phi _ t)
  | plam _ s => plam' _ (f_Phi _ s)
  | psubst _ s t => papp' _ (psubst' _ (f_Phi _ s)) (f_Phi _ t)
end.
```

O conjunto de termos da gramática auxiliar é construído tomado como base um conjunto `Phi'`, onde se define uma função **f** de `Phi` em `Phi'`. Dado que obtivemos uma prova de que **f** é injetiva, trabalhamos na construção de uma bijeção entre `Phi` e f(`Phi`), associando os elementos desses dois conjuntos de acordo com a seguinte lista:

Phi		f(Phi)
$\bullet$	$\longleftrightarrow$	$\bullet$
x	$\longleftrightarrow$	x
$(s\,t)$	$\longleftrightarrow$	$(s\,t)$
$\lambda x.s$	$\longleftrightarrow$	$\lambda x.s$
$s[x/t]$	$\longleftrightarrow$	$(s[x]\,t)$

Figura 4.15: *Bijeção entre* `Phi` *e* f(`Phi`).

Exemplo 4.3.2. *Segue um exemplo de como o procedimento de recursão é executado no termo* $y[y/(x\,x)]\{x/(z\,z)\}$*:*

$$\begin{array}{rcl} y[y/(x\,x)]\{x/(z\,z)\} & \xrightarrow{f} & (y[y]\,(x\,x))\{x/(z\,z)\} \\ & := & (y[y]\{x/(z\,z)\}\,(x\,x)\{x/(z\,z)\}) \\ & := & (y[y]\,(z\,z)) \\ & \xrightarrow{f^{-1}} & y[y/(z\,z)] \end{array}$$

Construímos uma função $\mathbf{f}^{-1}$: `Phi'` $\rightarrow$ `Phi`, a qual provamos ser o inverso de **f**, no domínio f(`Phi`) $\subset$ `Phi'`. Esse fato é demostrado através de indução sobre `Phi`. De posse dessas duas funções (**f** e $\mathbf{f}^{-1}$), definimos a meta-substituição no cálculo λex como sendo: $u\{x/b\} := \mathrm{f}^{-1}(\mathrm{f}(u)\{x/\mathrm{f}(b)\}')$.

```
Definition meta_subst (y : tmvar) (s : tm) (t : tm) : tm :=
f_inv (meta_subst' y (f_Phi s) (f_Phi t)).
```

Figura 4.16: *Meta-substituição no* λex*, via da gramática auxiliar.*

A função recursiva da meta-substituição para o cálculo λex é então definida, e conseguimos demonstra que ela possui, de fato, o comportamento esperado. Isso é feito através da formalização de cinco Teoremas, um para cada construtor do cálculo. Por exemplo, no caso da meta-substituição aplicada a substituição explícita, demonstramos que:

```
forall x y s t u, x <> y -> ~ In x (fvar s) ->
meta_subst y s (subst x t u) =
subst x (meta_subst y s t) (meta_subst y s u).
```

que na notação que criamos é equivalente a:

```
forall x t u, x <> y -> ~ In x (fvar s) ->
t^[x | u]^{x := s} = t^{x := s}^[x | u^{x := s}]
```

Capítulo 5

Propriedades do λex

Em [19], foi demonstrado, de maneira construtiva, que o cálculo λex possui importantes propriedades como C e MC, FC, SIM e a PSN (fato que é exibido na tabela da Figura 4.12. Aqui será dada especial atenção a demonstração das propriedades relativas a terminação do λex, ou seja, FC, SIM e a PSN.

```
Lemma basic_prop_1_rel_lex :  forall (s t :tm),
(s ->_lex t) -> Subset (fvar t) (fvar s).

Lemma basic_prop_2_lex :  forall x s t u,
(s ->_lex t) -> ((u ^{x := s}) ->_lex* (u ^{x := t})).

Lemma basic_prop_3_rel_lex :  forall x s t u,
(s ->_lex t) -> ((s ^{x := u}) ->_lex (t ^{x := u})).

Lemma basic_prop_4_rel_lex :  forall x t u,
SN rel_lex (t ^{x := u}) -> SN rel_lex t.
```

Figura 5.1: *Propriedades básicas.*

Lema 5.0.3 (Propriedades básicas)**.** *Seja $\mathcal{R} \in \{\mathtt{ex}, \lambda\mathtt{ex}\}$, e sejam t, t', u termos.*

(i) Se $t \to_{\mathcal{R}} t'$, então $\mathrm{fv}(t') \subseteq \mathrm{fv}(t)$;

(ii) Se $t \to_{\mathcal{R}} t'$, então $u\{x/t\} \xrightarrow{*}_{\mathcal{R}} u\{x/t'\}$ e $t\{x/u\} \to_{\mathcal{R}} t'\{x/u\}$. Então em particular $t\{x/u\} \in \mathcal{SN}_{\mathcal{R}}$ implica $t \in \mathcal{SN}_{\mathcal{R}}$.

Demonstração. Demonstramos o item *(i)* por indução sobre a relação de redução $\mathcal{R}$, e a definição de fv; e na primeira parte do item *(ii)* aplicamos o Teorema de indução α-estrutural em u, e a definição da meta-substituição, pois:

- Se $u = x$, então segue trivialmente que: $u\{x/t\} = t \to_{\mathcal{R}} t' = u\{x/t'\}$;
- Se $u = y \neq x$, então $y\{x/t\} = y = y\{x/t'\}$ e portanto $y\{x/t\} \xrightarrow{*}_{\mathcal{R}} y\{x/t'\}$;

- Se $u = (u'\,v')$, então $(u'\,v')\{x/t\} = (u'\{x/t\}\,v'\{x/t\}) \xrightarrow{*}_{\mathcal{R}} (u'\{x/t'\}\,v'\{x/t'\})$, por hipótese de indução e $(u'\{x/t'\}\,v'\{x/t'\}) = (u'\,v')\{x/t'\}$;

- Se $u = \lambda x.u'$, então $(\lambda x.u')\{x/t\} =_{\alpha} (\lambda z.u'')\{x/t\} = \lambda z.u''\{x/t\} \xrightarrow{*}_{\mathcal{R}} \lambda z.u''\{x/t'\}$, por hipótese de indução. Mas $\lambda z.u''\{x/t'\} = (\lambda z.u'')\{x/t'\} =_{\alpha} (\lambda x.u')\{x/t'\}$;

- Se $y \neq x$ e $u = \lambda y.u'$, então $(\lambda y.u')\{x/t\} = \lambda y.u'\{x/t\} \xrightarrow{*}_{\mathcal{R}} \lambda y.u'\{x/t'\}$, por hipótese de indução e $\lambda z.u'\{x/t'\} = (\lambda z.u')\{x/t'\}$.

- Se $u = u'[x/v']$, então $u'[x/v']\{x/t\} =_{\alpha} u''[z/v']\{x/t\} = u''\{x/t\}[z/v'\{x/t\}] \rightarrow_{\mathcal{R}} u''\{x/t'\}[z/v'\{x/t'\}]$, por hipótese de indução. Mas $u''\{x/t'\}[z/v'\{x/t'\}] = u''[z/v']\{x/t'\} =_{\alpha} u'[x/v']\{x/t'\}$;

- Se $y \neq x$ e $u = u'[y/v']$, então $u'[y/v']\{x/t\} = u'\{x/t\}[y/v'\{x/t\}] \xrightarrow{*}_{\mathcal{R}} u'\{x/t'\}[y/v'\{x/t'\}]$, por hipótese de indução. Mas $u'\{x/t'\}[y/v'\{x/t'\}] = u'[y/v']\{x/t'\}$.

E se, por exemplo $y \neq x$ e $t = (\lambda y.s)\,v \rightarrow_B s[y/v] = t'$, então:
$t\{x/u\} = ((\lambda y.s)\,v)\{x/u\} = ((\lambda y.s)\{x/u\}\,v\{x/u\}) = ((\lambda y.s\{x/u\})\,v\{x/u\})$
$\Rightarrow t\{x/u\} = ((\lambda y.s)\,v)\{x/u\} \rightarrow_B s\{x/u\}[y/v\{x/u\}] = s[y/v]\{x/u\} = t'\{x/u\}$.

□

5.1 Composição completa

Lema 5.1.1 (Composição completa). *Sejam t e $u \in \mathcal{T}$. Então $t[x/u] \xrightarrow{+}_{\mathtt{ex}} t\{x/u\}$.*

Demonstração. Segue diretamente por aplicação do Teorema α-indução estrutural sobre t, juntamente com a equação `C`.

- Se $t = x$, então $x[x/u] \rightarrow_{\mathtt{Var}} u = x\{x/u\}$;

- Se $t = y \neq x$, então $y[x/u] \rightarrow_{\mathtt{Gc}} y = y\{x/u\}$;

- Se $t = (u'\,v')$, então $(u'\,v')[x/u] \rightarrow_{\mathtt{App}} (u'[x/u]\,v'[x/u])$, mas por hipótese de indução $(u'[x/u]\,v'[x/u]) \xrightarrow{+}_{\lambda\mathtt{ex}} (u'\{x/u\}\,v'\{x/u\})$ e $(u'\{x/u\}\,v'\{x/u\}) = (u'\,v')\{x/u\}$;

- Se $t = \lambda x.u'$, então $(\lambda x.u')[x/u] =_{\alpha} (\lambda z.u'')[x/u] \rightarrow_{\mathtt{Lamb}} \lambda z.u''[x/u]$, mas hipótese de indução $\lambda z.u''[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} \lambda z.u''\{x/u\} = (\lambda z.u'')\{x/u\}$ e $(\lambda z.u'')\{x/u\} =_{\alpha} (\lambda x.u')\{x/u\}$;

- Se $y \neq x$ e $t = \lambda x.u'$, então $(\lambda y.u')[x/u] \rightarrow_{\mathtt{Lamb}} \lambda y.u'[x/u]$, mas hipótese de indução $\lambda y.u'[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} \lambda y.u'\{x/u\} = (\lambda y.u')\{x/u\}$;

- Se $t = u'[x/v']$, então $u'[x/v'] =_{\alpha} u''[y/v']$, com $y \neq x$ e os demais passos são semelhantes ao próximo item;

- Se $y \neq x$ e $t = u'[y/v']$, temos dois casos:

$x \in \mathrm{fv}(v') \;\Rightarrow u'[y/v'][x/u] \to_{\mathtt{Comp}} u'[x/u][y/v'[x/u]] \xrightarrow{+}_{\lambda\mathtt{ex}} u'\{x/u\}[y/v'\{x/u\}]$,
por hipótese de indução e $u'\{x/u\}[y/v'\{x/u\}] = u'[y/v']\{x/u\}$;

$x \notin \mathrm{fv}(v') \;\Rightarrow u'[y/v'][x/u] =_{\alpha} u''[z/v'][x/u]$, onde $z \notin \mathrm{fv}(u)$ e portanto
$u''[z/v'][x/u] =_{\mathtt{c}} u''[x/u][z/v'] \xrightarrow{+}_{\lambda\mathtt{ex}} u''\{x/u\}[z/v']$, por hipótese de indução.
Mas $u''\{x/u\}[z/v'] = u''\{x/u\}[z/v'\{x/u\}] = u''[z/v']\{x/u\} =_{\alpha} u'[y/v']\{x/u\}$.

□

Exemplo 5.1.2. *A título de exemplo, fornecemos a seguir uma comparação entre código da formalização do Lema 5.1.1 e sua prova em papel e lápis:*

<table>
<tr><td><code>Lemma full_comp:forall x t u,</code>
<code>t ^[x | u] ->_ex+ (t ^{ x := u}).</code></td><td>Sejam x variável, t e $u \in \mathcal{T}$,
então $t[x/u] \xrightarrow{+}_{\mathtt{ex}} t\{x/u\}$.</td></tr>
<tr><td><code>Proof.</code>
<code> intros x t u. pattern t.</code>
<code> apply tm_induction'.</code></td><td>indução α-estrutural sobre t</td></tr>
<tr><td><code>(* dot *)</code>
<code>autorewrite with lex.</code>
<code>assert (~ In x (fvar dot)).</code>
<code>rewrite fvar_dot. apply empty_neg_intro.</code>
<code>auto with lex.</code></td><td>$x \notin \emptyset = \mathrm{fv}(\bullet)$
$\bullet[x/u] \to_{\mathtt{Gc}} \bullet\; = \bullet\{x/u\}$</td></tr>
<tr><td><code>(* var *)</code>
<code>intro a. case (atom_eqdec _ a x).</code>
<code> (* a = x *)</code>
<code> intro H. autorewrite with lex.</code>
<code> rewrite H. auto with lex.</code>
<code> (* a <> x *)</code>
<code> intro H. autorewrite with lex.</code>
<code> assert (~ In x (fvar (var a))).</code>
<code> rewrite fvar_var.</code>
<code> apply singleton_neg_intro.</code>
<code> intro H'. apply sym_eq in H'.</code>
<code> contradiction.</code>
<code> auto with lex.</code></td><td>$x[x/u] \to_{\mathtt{Var}} u\; = x\{x/u\}$
$a \neq x \to a \notin \{x\} = \mathrm{fv}(x)$
$a[x/u] \to_{\mathtt{Gc}} a\; = a\{x/u\}$</td></tr>
<tr><td><code>(* app *)</code>
<code>intros s Hs t' Ht'.</code>
<code>autorewrite with lex.</code>
<code>apply Rplus_transitive with</code>
<code>(y := s ^{ x := u} ! (t' ^[x | u])).</code>
<code>apply Rplus_transitive with</code>
<code>(y := s ^[x | u] ! (t' ^[x | u])).</code>
<code>auto with lex. auto with lex.</code>
<code>auto with lex.</code></td><td>$(u'\,v')[x/u] \to_{\mathtt{App}} (u'[x/u]\,v'[x/u])$
$u'[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} u'\{x/u\}$ $(H.I.)$
$v'[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} v'\{x/u\}$ $(H.I.)$
$(u'[x/u]\,v'[x/u]) \xrightarrow{+}_{\lambda\mathtt{ex}} (u'\{x/u\}\,v'\{x/u\})$</td></tr>
<tr><td><code>(* lam *)</code>
<code>exists (add x (fvar u)).</code>
<code>intros a H s Hs.</code>
<code>destruct_neg_add H H1 H1.</code>
<code>autorewrite with lex.</code>
<code>apply Rplus_transitive with</code>
<code>(y := (& a, s ^[x | u])).</code>
<code>auto with lex. auto with lex.</code></td><td>$y \notin \{x\} \cup \mathrm{fv}(u)$
$(\lambda y.u')[x/u] \to_{\mathtt{Lamb}} \lambda y.u'[x/u]$
$u'[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} u'\{x/u\}$ $(H.I.)$
$(\lambda y.u')[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} (\lambda y.u')\{x/u\}$</td></tr>
</table>

```
  (* subst *)
  exists (add x (fvar u)).
  intros a H s Hs t' Ht'.
  destruct_neg_add H H1 H1.
  autorewrite with lex.
  case (In_dec x (fvar t')).
  (* In x (fvar t') *)
   intro H3.
   apply Rplus_transitive with
   (y := s ^[x | u] ^[ a | (t' ^[x | u])]);
   auto with lex.
   apply Rplus_transitive with
   (y := s ^{x := u} ^[ a | (t' ^[x | u])]).
   auto with lex.  auto with lex.
  (* In x (fvar t') *)
   intro H3.
   rewrite equation_C; trivial.
   rewrite meta_subst_not_fv with (M := t');
   trivial.
   auto with lex.
Qed.
```

$y \notin \{x\} \cup \mathrm{fv}(u)$

$x \in \mathrm{fv}(v')$

$u'[y/v'][x/u] \to_{\mathtt{Comp}} u'[x/u][y/v'[x/u]]$

$u'[x/u][y/v'[x/u]] \xrightarrow{+}_{\lambda\mathtt{ex}} u'\{x/u\}[y/v'\{x/u\}]$ (*H.I.*)

$u'\{x/u\}[y/v'\{x/u\}] = u'[y/v']\{x/u\}$

$x \notin \mathrm{fv}(v')$

$u'[y/v'][x/u] =_{\mathtt{C}} u'[x/u][y/v']$

$u'[x/u][y/v'] \xrightarrow{+}_{\lambda\mathtt{ex}} u'\{x/u\}[y/v']$ (*H.I.*)

$u'\{x/u\}[y/v'] = u'\{x/u\}[y/v'\{x/u\}] = u'[y/v']\{x/u\}$

□

A especificação do cálculo λex que utilizamos, extendendo a abordagem nominal de [23], proporciona uma grande proximidade entre provas em papel e lápis e suas respectivas formalizações em Coq. O Exemplo 5.1.2 é testemunha desse fato, contudo é necessário tomar certo cuidado, as teorias e bibliotecas implicítas nessa prova é que viabilizam tal conforto nas formalizações. Se analisarmos de forma isolada a construção da gramática do cálculo, baseda no arquivo `Nominal_Logic`, a conexão entre as provas em papel e as formalizações não é de forma alguma trivalmente aparente.

5.2 Simulação de um passo da β-redução

Lema 5.2.1 (Simulação de um passo da β-redução). *Sejam t e $t' \in \Lambda$. Se $t \to_\beta t'$, então $t \xrightarrow{*}_{\lambda\mathtt{ex}} t'$.*

Demonstração. $t = (\lambda y.s)\, v \to_\beta s[y/v] = t'$, mas $t = (\lambda y.s)\, v \to_B s[y/v]$ e $s[y/v] \xrightarrow{+}_{\mathtt{ex}} s\{y/v\} = t'$, pelo Lema da seção anterior. Portanto $t \xrightarrow{+}_{\mathtt{ex}} t'$, e como $\xrightarrow{+}_{\mathtt{ex}} \subset \xrightarrow{+}_{\lambda\mathtt{ex}} \subset \xrightarrow{*}_{\lambda\mathtt{ex}}$, então $t \xrightarrow{*}_{\lambda\mathtt{ex}} t'$ □

```
Lemma beta_one_step:  forall (t s:  tm),
(t ->_L s) -> (t ->_lex* s).
Proof.
 intros.  elim H.
 elim H0.  intros.
 assert ((& x, s0) !  t0 ->_lex* (s0^ [ x | t0 ])).
 auto with lex.  assert ((s0 ^[ x | t0]) ->_lex+ (s0 ^{ x := t0})).
 apply red_ex_lex_plus.  apply full_comp.
 apply Rplus_Rstar in H2.
 apply star_trans with (y := (s0 ^[ x | t0])); trivial.
Qed.
```

A demonstração do Lema 5.2.1 em Coq é, de fato, bastante curta, por que ela é feita por indução na relação de redução $\to_\beta$ (notacionada por `->_L`), somente através da aplicação de alguns resultados já formalizados, como o Lema 5.1.1 e lemas de inclusões e transitividade das relações de redução, que são importados dos arquivos `Reduct_Lex` e `sur_les_realations`.

5.3 Perpetualidade

Definiu-se uma estratégia perpétua para o λex da seguinte forma: se $t = xt_1...t_n$, reescreve-se o t_i mais a esquerda que é redutível. Se $t = \lambda x.u$, reescreve-se u; se $t = (\lambda x.s)u\overline{v_n}$, reescreve-se a cabeça do *redex*. Se $t = s[x/u]\overline{v_n}$, e $u \in \mathcal{SN}_{\lambda_{ex}}$, reescreve-se a cabeça do redex utilizando-se composição completa. Formalmente:

Definição 5.3.1 (Estratégia **perpétua** $\rightsquigarrow$)**.** *:*

$$\frac{\overline{u_n} \in \mathcal{NF}_{\lambda ex} \;\; t \rightsquigarrow t'}{x\overline{u_n}t\overline{v_m} \rightsquigarrow x\overline{u_n}t'\overline{v_m}}(\texttt{p-var}) \qquad \frac{t \rightsquigarrow t'}{\lambda x.t \rightsquigarrow \lambda x.t'}(\texttt{p-abs}) \qquad \frac{}{(\lambda x.t)u\overline{u_n} \rightsquigarrow t[x/u]\overline{u_n}}(\texttt{p-B})$$

$$\frac{u \in \mathcal{SN}_{\lambda_{ex}}}{t[x/u]\overline{u_n} \rightsquigarrow t\{x/u\}\overline{u_n}}(\texttt{p-subs1}) \qquad \frac{u \notin \mathcal{SN}_{\lambda_{ex}} \;\; u \rightsquigarrow u'}{t[x/u]\overline{u_n} \rightsquigarrow t[x/u']\overline{u_n}}(\texttt{p-subs2})$$

A estratégia é determinística, tal que se $t \rightsquigarrow u$ e $t \rightsquigarrow v$, então $u = v$. E além disso, ela não é necessariamente *leftmost-outermost* ou *left-to-right*, pois a regra `(p-subst1)` permite a propagação da substituição em qualquer ordem. Na sequência, os primeiros resultados obtidos são:

Lema 5.3.2. *Se $t \rightsquigarrow t'$, então $t \xrightarrow{+}_{\lambda\mathtt{ex}} t'$.*

Demonstração. O lema é provado por indução na estratégia $\rightsquigarrow$ e aplicação dos lemas 5.0.3 e 5.1.1:

(i) `(p-var)` $x\overline{u_n}t\overline{v_m} \rightsquigarrow x\overline{u_n}t'\overline{v_m}$, por hipótese de indução $t \xrightarrow{+}_{\lambda\mathtt{ex}} t'$ e portanto $x\overline{u_n}t\overline{v_m} \xrightarrow{+}_{\lambda\mathtt{ex}} x\overline{u_n}t'\overline{v_m}$;

(ii) `(p-abs)` $\lambda x.t \rightsquigarrow \lambda x.t'$, por hipótese de indução $t \xrightarrow{+}_{\lambda\mathtt{ex}} t'$ e portanto $\lambda x.t \xrightarrow{+}_{\lambda\mathtt{ex}} \lambda x.t'$;

(iii) `(p-B)` $(\lambda x.t)u\overline{u_n} \rightsquigarrow t[x/u]\overline{u_n}$. Temos que $(\lambda x.t)u \to_{\mathtt{B}} t[x/u]$ e portanto $(\lambda x.t)u\overline{u_n} \xrightarrow{+}_{\lambda\mathtt{ex}} t[x/u]\overline{u_n}$;

(iv) `(p-subst1)` $t[x/u]\overline{v_n} \rightsquigarrow t\{x/u\}\overline{v_n}$. Temos que $t[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} t\{x/u\}$, pela aplicação do Lema 5.1.1. Portanto $t[x/u]\overline{v_n} \xrightarrow{+}_{\lambda\mathtt{ex}} t\{x/u\}\overline{v_n}$;

(v) `(p-subst2)` $t[x/u]\overline{v_n} \rightsquigarrow t[x/u']\overline{v_n}$, por hipótese de indução $u \xrightarrow{+}_{\lambda\mathtt{ex}} u'$. Então pelo Lema 5.0.3 $t[x/u] \xrightarrow{+}_{\lambda\mathtt{ex}} t[x/u']$ e logo $t[x/u]\overline{v_n} \xrightarrow{+}_{\lambda\mathtt{ex}} t[x/u']\overline{v_n}$.

□

Assim como no Lema 5.2.1, a formalização de 5.3.2 utiliza resultados previamente demostrados no arquivo `Reduct_Lex`, como o Lema `mult_app_plus`, que afirma que: $t \to_{\lambda ex} t' \Rightarrow (t\, \bar{u_n}) \to_{\lambda ex} (t'\, \bar{u_n})$; além do Lema 5.1.1.

```
Lemma perp_1 :  forall a b, (a ->_$ b) -> (a ->_lex+ b).
Proof.
 intros a b H; elim H;
 intros a b H; elim H;
 intros; try apply mult_app_plus;
 auto with lex.
 apply red_ex_lex_plus.
 apply full_comp.
Qed.
```

Figura 5.2: *Compatibiliade da estratégia perpétua com a relação* `lex+`*.*

Lema 5.3.3 (Propriedade **IE**)**.** $u \in \mathcal{SN}_{\lambda ex}$ *e* $t\{x/u\} \in \mathcal{SN}_{\lambda ex}$ *então* $t[x/u] \in \mathcal{SN}_{\lambda ex}$.

```
Axiom ie_property :  forall x t u,
(SN rel_lex u) -> (SN rel_lex (t ^{x := u})) ->
(SN rel_lex (t ^[x | u])).
```

Figura 5.3: *Propriedade* IE*.*

Por hora, admitiremos o Lema 5.3.3 sem demostração. O artigo [19] fornece, através de uma técnica sofisticada de coloração das substituições explícitas, um capítulo completo a parte, para demonstração de tal propriedade. A dificuldade da formalização de tal propriedade reside no fato de que, a prova em papel e lápis considera uma extensão da gramática do λex (com diferentes tipos de substituições explícitas), o que a rigor tem que ser construído em Coq. Pretendemos concluir primeiramente os demais resultados relativos a terminação, e então concentraremos esforços na formalização da propriedade **IE**.

```
Theorem perpetuality :  forall t t’,
(t ->_$ t’) -> (SN rel_lex t’) -> (SN rel_lex t).
```

Figura 5.4: *Teorema da perpetualidade.*

Teorema 5.3.4 (Teorema da Perpetualidade)**.** *Sejam* $t, t' \in \mathcal{T}$ *tal que,* $t \rightsquigarrow t'$*. Se* $t' \in \mathcal{SN}_{\lambda ex}$ *então* $t \in \mathcal{SN}_{\lambda ex}$*.*

Demonstração. Lembrando que $l_{\mathcal{R}}(s) = max\{n \,|\, \exists s' \in \mathcal{NF}_{\mathcal{R}} : s \xrightarrow{n}_{\mathcal{R}} s'\}$. Então, efetuando-se indução sobre $\rightsquigarrow$, temos:

(i) `(p-var)` $t = x\overline{u_n}s\overline{v_m} \rightsquigarrow x\overline{u_n}s'\overline{v_m} = t'$, $\overline{u_n} \in \mathcal{NF}_{\lambda ex}$ e $s \rightsquigarrow s'$.
Se $\overline{u_n}s'\overline{v_m} \in \mathcal{SN}_{\lambda ex}$, então $v_m, s' \in \mathcal{SN}_{\lambda ex}$.
Portanto por hipótese de indução $s \in \mathcal{SN}_{\lambda ex}$ e logo $x\overline{u_n}s\overline{v_m} \in \mathcal{SN}_{\lambda ex}$;

(ii) `(p-abs)` $t = \lambda x.s \rightsquigarrow \lambda x.s' = t'$, e $s \rightsquigarrow s'$.
Se $\lambda x.s' \in \mathcal{SN}_{\lambda\mathtt{ex}}$, então $s' \in \mathcal{SN}_{\lambda\mathtt{ex}}$.
Mas por hipótese de indução $s \in \mathcal{SN}_{\lambda\mathtt{ex}}$ e logo $\lambda x.s \in \mathcal{SN}_{\lambda\mathtt{ex}}$;

(iii) `(p-B)` $t = (\lambda x.s)u\overline{u_n} \rightsquigarrow s[x/u]\overline{u_n} = t'$. Se $s[x/u]\overline{u_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$,
então $s, u, \overline{u_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$. Então deve-se mostrar, por indução sobre
$l_{\lambda\mathtt{ex}}(s)+l_{\lambda\mathtt{ex}}(u)+\sum_i l_{\lambda\mathtt{ex}}(u_i)$, que toda $\lambda\mathtt{ex}$-redução de $(\lambda x.s)u\overline{u_n}$ pertence a $\mathcal{SN}_{\lambda\mathtt{ex}}$.
Conclui-se que $(\lambda x.s)u\overline{u_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$;

(iv) `(p-subst1)` $t = s[x/u]\overline{v_n} \rightsquigarrow s\{x/u\}\overline{v_n} = t'$ e $u \in \mathcal{SN}_{\lambda\mathtt{ex}}$. Então pelo Lema 5.3.3
e pela hipótese de que $s\{x/u\}\overline{v_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$, conclui-se que $s[x/u], \overline{v_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$
e portanto $s[x/u]\overline{v_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$;

(v) `(p-subst2)` $t = s[x/u]\overline{v_n} \rightsquigarrow s[x/u']\overline{v_n} = t', u \notin \mathcal{SN}_{\lambda\mathtt{ex}}$ e $u \rightsquigarrow u'$.
Se $s[x/u']\overline{v_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$, então em particular $u' \in \mathcal{SN}_{\lambda\mathtt{ex}}$, e assim $u \in \mathcal{SN}_{\lambda\mathtt{ex}}$,
por hipótese de indução. Dado que $u \notin \mathcal{SN}_{\lambda\mathtt{ex}}$ e $u \in \mathcal{SN}_{\lambda\mathtt{ex}}$,
pode-se concluir qualquer proposição construtiva, em especial $t \in \mathcal{SN}_{\lambda\mathtt{ex}}$

□

A correta definição do conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$ exerce influência de forma direta na formalização do Teorema 5.3.4. Adaptamos a definição do predicado `SN` realizada na biblioteca *CoLoR* [26, 27], para definir construtivamente o conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$. Mas, não conseguimos ainda avançar na demostração de fatos simples sobre esse predicado, como por exemplo a afirmação de que: qualquer subtermo de um termo fortemente normalizável é também fortemente normalizável. Necessitamos da prova de tais lemas para que possamos conluir a formalização de 5.3.4.

Esse teorema fornece a garantia de que a definição da estratégia 5.3.1 é perpétua. Uma caracterização indutiva do conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$ poderá ser obtida, utilizando-se a estratégia perpétua definida. Esse tipo de caracterização é usualmente útil no desenvolvimento de provas de normalização, como é realizado na definição indutiva de $\mathcal{SN}_\beta$, que dada no Lema 3.4.5. Tal caracterização foi estendida para cálculos com substituições explícitas em [44, 45], mas nesses trabalhos são consideradas muitas regras de inferência para se caracterizar termos fortemente normalizáveis da forma $t[x/u]$. [19] obteve uma caracterização indutiva de $\mathcal{SN}_{\lambda\mathtt{ex}}$ onde é necessário somente uma regra para cada objeto da sintaxe do cálculo λex.

Definição 5.3.5 (Definição indutiva do conjunto $\mathcal{ISN}$)**.**

$$\frac{t_1, ..., t_n \in \mathcal{ISN} \quad n \geq 0}{x t_1 ... t_n \in \mathcal{ISN}}(\mathtt{var}) \qquad \frac{u[x/v]t_1...t_n \in \mathcal{ISN} \quad n \geq 0}{(\lambda x.u)v t_1...t_n \in \mathcal{ISN}}(\mathtt{app})$$

$$\frac{u\{x/u\}t_1...t_n \in \mathcal{ISN} \quad v \in \mathcal{ISN} \quad n \geq 0}{u[x/v]t_1...t_n \in \mathcal{ISN}}(\mathtt{subs}) \qquad \frac{u \in \mathcal{ISN}}{\lambda x.u \in \mathcal{ISN}}(\mathtt{abs})$$

Proposição 5.3.6. *$\mathcal{SN}_{\lambda\mathtt{ex}} = \mathcal{ISN}$.*

Demonstração. A demonstração é realizada com base na indução do par $<l_{\lambda\mathtt{ex}}(t), L(t)>$ e no Teorema 5.3.4:

- $t \in \mathcal{SN}_{\lambda\mathtt{ex}} \Rightarrow t \in \mathcal{ISN}$ é demonstrado por indução sobre $<l_{\lambda\mathtt{ex}}(t), L(t)>$;
- $t \in \mathcal{ISN} \Rightarrow t \in \mathcal{SN}_{\lambda\mathtt{ex}}$ é demonstrado por indução sobre t e aplicação do Teorema 5.3.4.

□

```
Lemma ISN_prop :  forall t, SN rel_lex t <-> ISN t.
```

Figura 5.5: *Caracterização indutiva do conjunto* $\mathcal{SN}_{\lambda\mathtt{ex}}$.

A demostração do Lema 5.3.6 faz uso do Teorema 5.3.4, o qual não conluimos a formalização. E mesmo admimitindo o 5.3.4 sem demonstração, seria ainda necessário formalizar o princípio de indução no par $<l_{\lambda\mathtt{ex}}(t), L(t)>$, que é utilizado na demonstração em papel e lápis de 5.3.6. Como ainda não obtivemos tais resultados, consideramos esse trabalho como pertencente a um escopo futuro de realização.

Teorema 5.3.7 (**PSN** para o λex)**.** *Se* $t \in \mathcal{SN}_{\beta}$*, então* $t \in \mathcal{SN}_{\lambda\mathtt{ex}}$.

Demonstração. A prova é realizada por indução na definição do conjunto $t \in \mathcal{SN}_{\beta}$ (Lema 3.4.5), utilizando-se a proposição 5.3.6.

- Se $t = x\overline{t_n}$ com $t_i \in \mathcal{SN}_{\beta}$, então $t_i \in \mathcal{SN}_{\lambda\mathtt{ex}}$ por hipótese de indução, e então por aplicação da regra `(var)` conclui-se que $x\overline{t_n} \in \mathcal{ISN} = \mathcal{SN}_{\lambda\mathtt{ex}}$;
- Se $t = \lambda x.s$ com $s \in \mathcal{SN}_{\beta}$, então $s \in \mathcal{SN}_{\lambda\mathtt{ex}}$ por hipótese de indução. Logo por aplicação da regra `(abs)`, é possível concluir que $\lambda x.s \in \mathcal{SN}_{\lambda\mathtt{ex}}$
- Se $t = (\lambda x.u)v\overline{t_n}$ com $u\{x/v\}\overline{t_n}$, $v \in \mathcal{SN}_{\beta}$. Então ambos os termos pertencem a $\mathcal{SN}_{\lambda\mathtt{ex}}$ por hipótese de indução. Portanto ao se aplicar a regra `(subs)` obtém-se que $u[x/v]\overline{t_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$, e logo ao se aplicar a regra `(app)` se chega a conclusão de que $(\lambda x.u)v\overline{t_n} \in \mathcal{SN}_{\lambda\mathtt{ex}}$.

□

```
Theorem PSN : forall t, SN rel_lambda t -> SN rel_lex t.
```

Figura 5.6: *Preservação da normalização forte.*

Como é possível observar na demostração de 5.3.7, tal teorema é um corolário da proposição 5.3.6. Assim, objetivamos concluir a formalização de tal proposição, antes de abordar o 5.3.7, que de certo modo, conclui uma cadeia de formalizações.

Capítulo 6

Conclusão e trabalhos futuros

A tarefa de verificação formal de uma teoria é algo desafiador e estimulante. Pois a construção de um conjunto de ferramentas para verificações formais, que aproxime cada vez mais provas mecânicas de provas realizadas em papel e lápis, é uma busca global, que envolve um número grande de pesquisadores ao redor do mundo [46]. Nosso trabalho, teve como foco a formalização do cálculo de substituições λex, e suas propriedades de normalização.

Uma questão importante, é a escolha de um determinado assitente de prova para realização de formalizações, que é algo de cunho mais pessoal, do que técnico. No entanto, nossa opção pelo assistente Coq levou também em consideração o tipo de teoria com o qual esse sistema trabalha, que é o CCI (Cálculo de Contruções Indutivas). A teoria contrutiva é uma restrição da lógica clássica, e assim a realização de formalizações nesse contexto, insere uma certa dificuldade, dado que alguns axiomas, como a *dupla negação* e a *lei do terceiro excluído*, não podem ser utilizados.

Em contrapartida, a extração de código excutável, para a obtenção de programas certificados, parece ser mais simples em um assitente baseado em uma teoria construtiva. *A priori*, esse trabalho não objetivou a obtenção de códigos de programas executáveis, mas temos a esperança de que nossa biblioteca de formalizações do cálculo λex, assim como qualquer outra elaborada em um ambiente construtivo, possa ser também utilizada para tal fim.

Alguns dos exemplos mais conhecidos de cálculos de substituições explícitas, designam suas variáveis através da notação de índices de "de Bruijn". Esse tipo de notação nasceu para realizar a tarefa de construção de linguagens de programação, no ambiente em que uma máquina pode interpretar com rapidez e agilidade os termos da gramática do cálculo. A grande vantagem dessa abordagem é que ela dispensa o uso de α-equivalência entre termos, dado que um objeto do tipo $\lambda\underline{1}$ representa uma classe de equivalência $a \in \Lambda/_{=_\alpha}$.

Em um primeiro trabalho de iniciação científica, realizamos uma formalização do cálculo $\lambda\sigma$, que é um exemplo muito conhecido de cálculos de substituições explícitas, que utilizam notação de "de Bruijn". E quando realizamos o projeto de formalização de um cálculo com *nomes*, o λex, nos deparamos com o trabalho de Délia Kesner [19]: *Perpetuality for full and safe composition (in a constructive setting)*; e nos sentimos, de certo

modo, seduzidos pela característica construtiva das provas realizadas, pela simplicidade de notação e do conjunto de regras de reescrita, aliados ao poder do cálculo, que possui muitas das propriedades esperadas para uma extensão do cálculo λ.

Contudo a designação de variáveis por nomes tem seu preço. No Coq definições indutivas não consideram ligações de variáveis, então qualquer construtor é necessariamente injetivo, isto é, dado $K : \sigma \to \theta$ um construtor e $x_1, x_2 : \sigma$, $K\, x_1 = K\, x_2 \to x_1 = x_2$. Isso é um problema quando definimos indutivamente a gramática de um cálculo com nomes, pois nos construtores com variáveis ligadas, que no nosso caso são a abstração e a substituição explícita, não há injetividade com respeito a relação =, pré-contruída no Coq: $\lambda x.a = \lambda y.b \nrightarrow x = y \wedge a = b$ e $s[x/v] = t[y/v] \nrightarrow s = t \wedge x = y$. Há dois caminhos a seguir, a primeira opção seria a construção de uma relação $=_\alpha$ não contida em =; a segunda é não fazer uso da definição do indutiva do Coq e para definição da gramática do cálculo, e considerar $=_\alpha$ contida em =. A especificação de *Aydemir et al* [23], a qual tomamos como base, adotou a segunda opção.

Essa questão, de formalização de sintaxes abstratas com nomes e variáveis ligadas, surge primeiramente em um contexto em que se fazia uso do assistente de prova *Isabelle / HOL* [4]. Tal trabalho é apresentado por Christian Urban [34] e ele gerou motivações para que formalizações de sintaxes com ligações fossem estendidas a outros assistentes de prova, como o Coq. O artigo [23], de B. Aydemir *et al*, apresenta uma biblioteca de Lógica Nominal [20, 21, 22] (que se encontra no arquivo `Nominal_Logic`[1]), e formaliza a gramática do cálculo λ. Essa formalização fornece os princípios de recursão e indução α-estrutural [24] do λ.

Estendemos a especificação de [23] incluindo a substituição explícita (s[x/t]) na gramática definida, e formalizamos o princípio de indução α-estrutural para a gramática do λex. Construímos assim uma gramática auxiliar, a qual chamamos de λex', e nessa sintaxe conseguimos formalizar ambos os teoremas de indução e recursão α-estrutural. Ao criar uma bijeção entre as gramáticas do λex e λex', obtivemos como resultado a definição da operação de meta-substituição para o λex.

Definimos as relações de redução do λex, os fechos transitivo / reflexivo-transitivo das relações, utilizando a biblioteca `sur_les_relations` [25]. Formalizamos resultados com respeito a compatibilidade entre as relações redução e a operação de meta-substituição. Concluímos com a demostração dos Lemas da Composição Completa, da Simulação de um passo da β-redução e a compatibilidade entre a estratégia de redução perpétua e a relação $\to_{\lambda\mathtt{ex}}$. Criamos também a definição do predicado `SN` (Fortemente Normalizável) utilizando a biblioteca *CoLoR* [26, 27].

Como trabalhos futuros, pretendemos concluir a formalização dos Teoremas da Perpetualidade, da Caracterização Indutiva do Conjunto $\mathcal{SN}_{\lambda\mathtt{ex}}$, e do Teorema da *Preservação da Normalização Forte*, assim como a formalização completa do Teorema de Recursão α-estrutural para o λex. A solução de criação de uma gramática auxiliar, para construção da função de meta-substituição no λex, surgiu com uma alternativa durante a realização das provas. Finalizando, consideramos também um outro ponto interessante a ser abordado: a questão de comparação, com respeito as propriedades de normalização, entre o cálculo λex e um cálculo baseado na gramática λex'.

[1] `http://www.cic.unb.br/~flavio/msc/lex.tar.gz`

Referências

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. 1, 3, 5, 36, 45

[2] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.2*. INRIA-Rocquencourt, 2008. 1, 2, 3, 6

[3] N. G. de Bruijn. The mathematical language automath, its usage and some of its extensions. *Symposium on Automatic Demonstration*, 125, 1970. 2

[4] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 2, 28, 57

[5] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer Verlag. 2

[6] A. Trybulec. The mizar-qc/6000 lógic information language. *ALLC Buletin*, 6(2):136–140, 1978. 2

[7] 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. 2

[8] H. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (revised edition)*. North Holland, 1984. 3, 6, 16, 25, 27

[9] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941. 3, 25

[10] D. Kesner. The theory of calculi with explicit substitutions revisited. In *Proc. 16th Annual Conference on Computer Science and Logic (CSL-07)*, LNCS. Springer-Verlag, 2007. 3, 5, 16, 41, 44, 45

[11] H. Barendregt, J. Bergstra, J. W. Klop, and H. Volken. Degrees, reductions and representability in the lambda calculus. *Technical Report Preprint 22, University of Utrecht, Departament of Mathematics*, 1976. 3

[12] J. A. Bergstra and J. W. Klop. Church-rosser strategies in the lambda calculus. *Theoretical Computer Science*, 9:27–38, 1979. 3

[13] J. A. Bergstra and J. W. Klop. Strong normalization and perpetual reductions in the lambda calculus. *Journal of Information Processing and Cybernetics*, 18:403–417, 1982. 3

[14] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958. 3

[15] J. W. Klop. Combinatory Reduction Systems. *Mathematical Center Tracts*, 27, 1980. 3

[16] J.-J Lévy. Optimal reductions in the lambda-calculus. *Academic Press Limited*, pages 159–191, 1980. 3

[17] R. C. de Vrijer. *Surjective Pairing and Strong Normalization*. PhD thesis, University of Amsterdam, 1987. 3

[18] F. van Raamsdonk, P. Severi, M. H. Sørensen, and H. Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999. 3, 33

[19] 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*, 2008. 3, 4, 16, 48, 53, 54, 56

[20] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999. 4, 57

[21] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003. 4, 57

[22] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001. 4, 57

[23] B. E. Aydemir, A. Bohannon, and S. Weirich. Nominal reasoning techniques in coq: (extended abstract). *Electr. Notes Theor. Comput. Sci.*, 174(5):69–77, 2007. 4, 5, 17, 21, 22, 26, 28, 32, 38, 39, 45, 51, 57

[24] A. M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, 2006. 4, 17, 23, 57

[25] A. Saibi. Formalization of a lamda-calculus with explicit substitutions in coq. In P. Dybjer, B. Nordström, and J. M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 1994. 4, 5, 14, 38, 57

[26] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR: a Coq Library on Rewriting and Termination. *http://color.inria.fr/*, 2006. 4, 16, 54, 57

[27] F. Blanqui and A. Koprowski. Automated Verification of Termination Certificates. Technical report, INRIA, 2009. 4, 16, 54, 57

[28] H. Barendregt. λ-calculi with types. *Handbook of Logic in Computer Science*, II, 1992. 6

[29] The Coq Club. The Coq users' Contributions. Available on the Internet, 2010. http://coq.inria.fr/contribs/. 6

[30] M. Ayala-Rincón and F. L. C. de Moura. Fundamentos da programação lógica e funcional - o principio da resolução e a teoria de reescrita. Notas de Aula, 2008. 13

[31] S. Kleene and B. Rosser. The inconsistency of certain formal logics. *Annals of Math.*, 36(2):630–636, 1935. 25

[32] A. Church and J. B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39:472–482, 1936. 25

[33] A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, revised edition, 1925-1927. Three volumes. The first edition was published 1910-1913. 25

[34] C. Urban. Nominal Techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008. 28, 57

[35] A. Charguéraud. The locally nameless representation. `http://arthur.chargueraud.org/research/2009/ln/`. Acessado em 21/06/10, 2010. 28

[36] 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. 36

[37] T. Hardin and J.-J. Lévy. A Confluent Calculus of Substitutions. *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989. 38, 41, 45

[38] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Remy, editors, *CTRS*, volume 656 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 1992. 39, 45

[39] R. Lins. A new formula for the execution of a categorical combinators. In *8th Conference on Automated Deduction (CADE)*, volume 230 of *LNCS*, pages 89–98. Springer-Verlag, 1986. 39, 45

[40] R. Lins. Partial categorical multi-combinators and church rosser theorems. Technical report, Computing Laboratory, University of Kent at Canterbury, 1992. 39, 45

[41] R. Bloo and K. H. Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *Computer Science in the Netherlands*, November 1995. 39, 45

[42] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *TLCA'95*, volume 902 of *LNCS*. Springer-Verlag, 1995. 41

[43] F. Kamareddine and A. Ríos. The λs-calculus: its typed and its extended versions. Technical report, Department of Computing Science, University of Glasgow, 1995. 45

[44] S. Lengrand, P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini, and S. van Bakel. Intersection types for explicit substitutions. *Information and Computation*, 189(1):17–42, 2004. 54

[45] E. Bonelli. *Substitutions explicites et réécriture de terms*. PhD thesis, Université Paris XI, 2001. 54

[46] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. *LNCS*, 3603:50–65, 2005. 56