Fabrício Sanches Paranhos

Uma Formalização da Teoria Nominal em Coq

Goiânia, Goiás

2021

Resumo

Apresentamos uma formalização em progresso de conjuntos nominais em Coq, salientando as principais decisões de design e projeto. Implementamos e comparamos dois métodos, um pragmático, apoiada nos pilares básicos da teoria nominal, e uma formalização construtiva de conjuntos nominais, propriamente dita. Mostramos que graças a classe de tipos e reescrita generalizada alcançamos definições e provas concisas, ao mesmo tempo evitando o conhecido cenário "*setoid hell*".

Palavras-chave: conjuntos nominais, técnicas nominais, formalização, Coq.

Sumário

1 Introdução

A representação de estruturas ligantes ainda é um problema em aberto na área de implementação de linguagens de programação e verificação formal de sistemas. Dentre as principais técnicas podemos citar: índices de de Bruijn (1), *locally nameless* (2), nominal (3) e abordagens de sintaxe abstrata de alta ordem (4, 5). Aydemir *et al.* (6) e Ambal (7) *et al.*, compararam formalizações, respectivamente do cálculo λ e do cálculo π, utilizando os métodos citados, entretanto nenhuma das técnicas se sobressai as demais. Os índices de de Bruijn tem a metateoria com implementação mais simples, de acordo com ambos os trabalhos, mas necessita da maior quantidade de lemas auxiliares, algo já notado por Hirschkoff em (8): do total de 800 lemas, 600 eram relacionados a operações de índices. *Locally nameless* melhora a situação dos índices, porém com a necessidade da adição de um predicado (filtro) de termo bem formado a todas as propriedades. Sintaxe abstrata de alta ordem utiliza funções da metalinguagem (assistente de provas) para representar ligantes, o que dificulta seu raciocínio, já que o assistente não pode discursar sobre propriedades de suas próprias abstrações. Já as técnicas nominais, de acordo com (7), é mais simples de escrever lemas e definições, já que são fieis às informais, ao custo de uma infraestrutura complexa, disponível apenas em Isabelle/HOL (6).

Dado a elaborada metateoria das técnicas e conjuntos nominais (9, 10), formalizações que utilizam de técnicas nominais (11, 12, 13, 14, 7) tendem a implementar apenas os aspectos fundamentais da teoria: permutação de nomes e ação de permutação. Pois ação de permutação e permutação de nomes em conjuntos possuem definições mais simples e apresenta um comportamento mais previsível do que a substituição padrão de termos, já que não depende da noção de variável livre ou ligada e não há como ocorrer captura de variável livre (15).

Especificamente em relação a implementações das técnicas nominais, para lógicas clássicas, o pacote nominal (16, 12) do assistente de provas Isabelle/HOL (17) é considerado estado da arte (18), visto que é um sistema completo e automatizado. Porém, para lógicas construtivas as opções são bem inferiores. Aydemir *et. al.* propôs uma prova de conceito em (11) em Coq, na qual uma ferramenta externa leria a descrição de uma sintaxe, para em seguida construir uma especificação nominal axiomatizada. Todavia supõem-se o abandono do método, em favor do *locally nameless*, ao analisar as publicações posteriores pelo grupo de pesquisa. Em Agda tem-se os trabalhos de Copello *et. al.* (13, 14) , no qual foram capazes de derivar um princípio de indução e um combinador de recursão α-estrutural (9), dependendo apenas de uma condição extra, que chamaram de α-compatibilidade (Definição 10). Infelizmente o método consta apenas com uma implementação manual para uma restrição do cálculo λ, com pouco espaço para automação no estado atual. Por último,

Choudhury em sua dissertação de mestrado sob orientação de Pitts (15), apresenta uma formalização construtiva de conjuntos nominais em Agda que, apesar de não ser aplicada, é o desenvolvimento formal mais completo de conjuntos nominais.

Pelo estado da arte atual da teoria nominal e sua aplicação em assistentes de provas construtivos, como observado por Andrew Pitts em entrevista no recebimento do prêmio Alonzo Church 2019 (19), propomos abordar a falta de uma biblioteca nominal, similar ao pacote nominal do Isabelle, no assistente de provas Coq. Neste trabalho apresentamos uma formalização em progresso de conjuntos nominais. Se as técnicas nominais permitem a formalização mais próxima das provas informais, gostaríamos de argumentar que, formalizar conjuntos nominais em Coq permite também alcançar desenvolvimentos sucintos da metateoria, similares à metateoria informal de conjuntos nominais (10), graças aos mecanismos de classes de tipos e reescrita generalizadas do Coq.

Este projeto está organizado da seguinte forma: o Capítulo 2 apresenta o referencial teórico de conjuntos nominais. Uma breve descrição do assistente de provas Coq, e dos mecanismos de classe (Seção 3.2) e reescrita (Seção 3.4) encontram-se no Capítulo 3. O Capítulo 4 apresenta nossas experimentações no desenvolvimento de uma biblioteca nominal: a Seção 4.2 apresenta uma implementação da metodologia de Copello *et. al.*, enquanto a Seção 4.3 contém a formalização em progresso de conjuntos nominais, similar a (15). E por último no Capítulo 5 apresentamos um cronograma com etapas futuras do projeto e nossas conclusões e perspectivas futuras. As formalizações em Coq estão disponíveis nos repositório do GitHub do autor (20, 21).

2 Referencial Teórico

Neste capítulo apresentamos os conceitos e definições fundamentais utilizados neste trabalho. Começamos pelo conceito de nome, seguido de permutações, pilares da teoria nominal (3, 22, 10), e finalizamos com conjuntos nominais, também chamados de tipos nominais por alguns autores (12, 15), incluindo o conceito de suporte.

Nomes são úteis por seus identificadores, assim como ponteiros em C/C++, ou referencias em Java ou C#. Sua estrutura interna é abstrata, irrelevante e indivisível. Por isso são denominadas átomos. Assumimos um conjunto contável[1] infinito de nomes, com igualdade decidível, denominado $\mathbb{A}$, representado pelas letras minúsculas no início do alfabeto: a, b, etc. A teoria nominal captura a noção de (in)dependência de nomes através de uma metateoria, simples e uniforme, baseada em permutações de nomes.

As permutações são funções bijetoras, de um conjunto para si mesmo, na qual formam um grupo de simetria, através da composição de funções ($\circ$) como operação binária. Grupo, é um conjunto G fechado sob uma operação binária $* : G \to G$, denotado por $\langle G, * \rangle$, que satisfaz as seguintes condições: a operação binária é associativa; existe um elemento $\varepsilon \in G$, chamado de neutro ou identidade, tal que para todo $g \in G$ temos $\varepsilon * g = g * \varepsilon = g$; e para cada $g \in G$ existe $g' \in G$, chamado de inversa, tal que $g * g' = g' * g = \varepsilon$ (23). O elemento inverso pode ser obtido via uma função $f : G \to G$, chamada de função inversa, dessa forma a última condição é alterada para $g * f(g) = f(g) * g = \varepsilon$.

Uma propriedade de permutações, é que estas podem ser decompostas em 2-ciclos, também chamados de transposição ou *swap*:

Definição 1 (Transposição ou *Swap*)**.** *Seja X um conjunto, tal que a $b \in X$. Transposições são permutações 2-ciclos, representado por $\langle a\ b \rangle : X \to X$ e definido como:*

$$\langle a\ b \rangle\ c \triangleq \begin{cases} b & \text{se } a = c \\ a & \text{se } b = c \\ c & \text{caso contrário} \end{cases} \tag{2.1}$$

Fazendo $X = \mathbb{A}$, obtemos a operação de transposição de nomes, essencialmente uma função que troca nomes. O interesse em utilizar transposição como operação básica de alteração sintática, em gramáticas com estruturas ligantes, ao contrário da substituição de termos, está no fato de não dependerem da definição de variável livre ou ligada e serem imunes a captura de variáveis livres (22).

[1] Não é estritamente necessário que o conjunto de nomes seja contável, veja (10, Exercício 6.2, página 109).

Denotamos por $Perm\mathbb{A} = (\mathbb{A}, \circ)$ o grupo de permutações finitas sobre $\mathbb{A}$, formado pela composição de transposições, com a função identidade como elemento neutro. Reservamos as letras minúsculas p, q, r e s para designar membros deste grupo. Permutações agem sobre conjuntos por meio da operação de ação (à esquerda) de grupo:

Definição 2 (Ação de grupo (10))**.** *Seja $(G, \circ)$ um grupo com elemento neutro representado por ε. Define-se ação à esquerda de G sobre um conjunto X qualquer como uma função $(\bullet) : G \times X \to X$, que satisfaz:*

$$\forall x \in X, \;\; \varepsilon \bullet x = x \tag{2.2}$$

$$\forall g \; h \in G, \forall x \in X, \;\; g \bullet (h \bullet x) = (g \circ h) \bullet x \tag{2.3}$$

Definição 3 (Ação de permutação e conjunto de permutação (10))**.** *Ação de permutação é uma ação definida pelo grupo $Perm\mathbb{A}$. Conjunto de permutação é aquele dotado de uma ação de permutação.*

Ao enriquecer um conjunto com uma ação de permutação (Definição 3) conseguimos estabelecer uma noção de dependência de nomes, onde o conjunto de nomes ao qual uma estrutura depende é chamado de **suporte**, enquanto seu complemento é denominado de nomes frescos. Mas primeiramente, vamos discutir o significado informal de (in)dependência, no contexto do cálculo λ, pois seu significado depende da gramática e semântica em questão.

O conjunto dos termos (Λ) do cálculo λ é dado pela gramática (24), utilizando nomes no lugar de variáveis:

$$M, N \in \Lambda ::= a \mid (MN) \mid (\lambda a.M) \tag{2.4}$$

onde estamos utilizando nomes no lugar de variáveis, ou seja $a \in \mathbb{A}$, (MN) representa a aplicação de dois termos e $(\lambda a.M)$ denota uma abstração. O conjunto de variáveis livres é definido como:

Definição 4 (Variáveis livres e ligadas (24))**.** *O conjunto de variáveis livres é definido recursivamente na estrutura de um termo pela função $FV : \Lambda \to \mathbb{A}$, para $a \in \mathbb{A}$ e $M, N \in \Lambda$:*

$$FV(a) = \{a\}, \;\; FV(MN) = FV(M) \cup FV(N), \;\; FV(\lambda a.M) = FV(M) - \{a\}$$

O conjunto de variáveis ligadas é definido recursivamente na estrutura de um termo pela função $BV : \Lambda \to \mathbb{A}$, da seguinte maneira:

$$BV(a) = \{\emptyset\}, \;\; BV(MN) = BV(M) \cup BV(N), \;\; BV(\lambda a.M) = BV(M) \cup \{a\}$$

Dizemos que termos são α-equivalentes, denotado por $M \sim_\alpha N$, se estes diferem apenas nos nomes das variáveis ligadas.

Um termo, por exemplo $W = \lambda x.(xy)$, depende de todas as variáveis ($\mathrm{BV}(W) \cup \mathrm{FV}(W)$) contidas nele, pois a modificação de qualquer nome implica num novo termo, sintaticamente diferente, como $\lambda x.(xz)$ ou $\lambda z.(zy)$. Entretanto, se considerarmos termos módulo α-equivalência, o conjunto de nomes dependentes são apenas as variáveis livres ($\mathrm{FV}(W)$), pois continuando com nosso exemplo: $\lambda x.(xy) \not\sim_\alpha \lambda x.(xz)$, mas $\lambda x.(xy) \sim_\alpha \lambda z.(zy)$. Formalmente definimos suporte como:

Definição 5 (Suporte)**.** *Seja X um conjunto de permutação, então $S \subset \mathbb{A}$ suporta $x \in X$ se:*

$$\forall p \in Perm\mathbb{A}, ((\forall a \in S, p \bullet a = a) \rightarrow p \bullet x = x) \tag{2.5}$$

Ou seja, se uma permutação p não modifica os nomes do suporte S de x, então essa mesma permutação não terá efeito em x. Ao contrário, qualquer p que modifique x modifica algum nome em seu suporte. A elegância da definição está no fato de fornecer uma forma simples de especificar dependência de nomes, apenas através da noção de permutação de nomes e a noção de suporte.

Gostaríamos de ressaltar dois pontos extras sobre a definição de suporte: (1) é importante que seja sempre finito. Caso $S = \mathbb{A}$, em suporte, então não é possível obter novos nomes frescos, pois esgotaram-se as opções. (2) Podem existir mais de um suporte (S_1 e S_2) para um único elemento. Nestes casos é possível mostrar que $S_1 \cap S_2$ também é um suporte (10, Proposição 2.3). O que nos leva a definição de menor suporte:

Definição 6 (Menor suporte)**.** *Seja X um conjunto de permutação, o menor suporte é dado pela intersecção de todos os suportes de $x \in X$:*

$$supp(x) \triangleq \bigcap \{S \mid S \text{ suporta } x\} \tag{2.6}$$

e frescor, até agora utilizado informalmente:

Definição 7 (Frescor)**.** *Dado dois conjuntos de permutação X e Y, dizemos que $y \in Y$ é fresco em $x \in X$, denotado por $y\#x$, se:*

$$supp(y) \cap supp(y) = \emptyset \tag{2.7}$$

Como na maioria dos casos $Y = \mathbb{A}$, temos que:

$$y\#x \leftrightarrow y \notin supp(x) \tag{2.8}$$

Assim, temos todas as ferramentas necessárias para definir conjuntos nominais:

Definição 8 (Conjunto nominal)**.** *Conjunto nominal é um conjunto de permutação na qual todos elementos possuem menor suporte.*

Apresentamos neste capítulo a teoria clássica de conjuntos nominais. No restante do texto começamos a explorar diferentes possibilidades de implementação de conjuntos nominais, em assistentes de provas construtivos, com teoria de tipos dependentes, mais especificamente no Coq, apresentado no Capítulo 3.

3 Metodologia

3.1 Coq

O assistente de provas Coq (25) é um sistema gerenciador de desenvolvimentos formais, composto por três linguagens: Gallina (especificação), Vernacular (comandos) e Ltac (metaprogramação). O vernacular define comandos que permitem interagir com o ambiente de provas, como adicionar definições, realizar consultas e alterar configurações do assistente. Ltac é utilizado para metaprogramação, sendo principalmente utilizada no desenvolvimento de táticas e automação. Por último, Gallina é uma linguagem funcional (de especificação) com tipos dependentes, na qual é implementado o Cálculo de Construções (Co)Indutivas (26, 27, 28). Para o desenvolvimento de provas, através do isomorfismo de Curry-Howard (29), Coq permite construir termos Gallina (programas), interativamente, por meio da aplicação de táticas. Ao final, o termo construído é enviado ao *kernel* do Coq para verificação. A seguir, destaco alguns tópicos mais relevantes ao desenvolvimento deste trabalho.

3.2 Classes de tipos

As classes de tipos em Coq, são semelhantes a implementação em Haskell (30), isto é, ambos um mecanismo de polimorfismo *ad-hoc* (31) em que funções podem ser aplicadas a diferentes argumentos, dependendo do seu tipo. No assistente de provas Coq, classes de tipos são açúcar sintático (*syntax sugar*) para registros paramétricos dependentes e contam, também, com um sistema de busca de provas e inferência, similar a Prolog. Por consequência, as classes de tipos são cidadãs de primeira classe, dessa forma restrições de classe são apenas parâmetros implícitos (32). Por exemplo, em Haskell, a classe de tipos ordenados depende da classe de tipos com igualdade (33), representado: `Eq a => Ord a`. Afim de obter o mesmo efeito em Coq, supondo a existência das mesmas classes, basta passarmos como parâmetros a classe **`Eq`** a classe **`Ord`**: `Ord (a: Type) (e: Eq a)`. Logo, classes de tipos em Coq são mais poderosas que sua contrapartida em Haskell.

Classes de tipos trazem também uma nova possibilidade, mais simples, para a construção de uma hierarquia de estruturas algébricas. Dentre diversas implementações de sucesso variado (34, 35, 36, 37). Spitters e van der Weegen (38) propõe justamente isto, definir uma hierarquia algébrica através de classes, com um detalhe: separação entre, no que eles chamam de *unbundling*, classe operacionais e classes predicados. Classes operacionais permitem referenciar operações, como operadores binários em grupos e monoides, concedendo um nome e uma notação canônica. Enquanto classes predicadas

agrupam propriedades, que no caso de grupos representam seus axiomas (veja seção 3.5).

3.3 Igualdade e *setoids*

A igualdade padrão do Coq é sintática, definida pelo tipo indutivo `eq` e representado pelo símbolo "=":

```
Inductive eq (A: Type) (x: A): A → Prop := eq_refl: x = x.
```

O código acima define uma família de igualdades parametrizada pelo tipo `A`. A única forma de construir uma prova `a = b` é através do construtor `eq_refl`, desde que `a` e `b` sejam intencionalmente, ou sintaticamente iguais (39). Igualdade intencional ocorre quando dois termos são convertíveis, isto é, são redutíveis entre sí, via regras de conversão do Coq: α, β, ι, δ, ζ e η-expansão (40). Esta definição é bem útil para reescrita, como pode ser visto no princípio de indução/recursão gerado para `eq`:

```
eq_rect: ∀ (A: Type) (x: A) (P: A → Type), P x → ∀ y: A, x = y → P y.
```

pois permite reescrever termos, em qualquer contexto `P`, desde que sejam intencionalmente iguais.

Especificações e formalizações podem apresentar objetos sintaticamente diferentes, não-convertíveis, mas que representam o mesmo objeto semanticamente, identificados por meio de uma relação de equivalência, por exemplo termos α-equivalentes do cálculo λ. Para estes casos, não podemos aplicar a família de táticas de reescrita do Coq, pois utilizam `eq_rect` em sua implementação, o que acaba restringindo seu uso apenas à igualdade sintática. Uma solução ingênua seria tentar definir conjuntos quocientes, entretanto não seria possível sem a adição de axiomas extras (41), além de tornar o algoritmo de checagem de tipos indecidível (34). A alternativa é utilizar *setoids*.

Os *setoids*, também conhecidos como conjuntos de Bishop (42, 43), são estruturas formadas por um tipo equipado com uma relação de equivalência, geralmente implementados por meio de registros:

```
Record Setoid (A: Type) := {
   car: A;
   eqv: relation A;
   prf: Equivalence equiv
}
```

O grande problema de *setoids* é que, na prática, o usuário tem que lidar constantemente com os detalhes da implementação. Por exemplo, supondo uma representação da classe de α-equivalência para termos-λ, denotada por `Λα`:

```
Definition Λα: Setoid Λ := {| car := Λ; eqv: α; prf := _ |}.
```

com os símbolos `Λ` e `α` representando, respectivamente, o conjunto dos termos-λ e a relação

de α-equivalência. Uma função que extrai o corpo da abstração-λ, representado pelo construtor `Lam`, pode ser definido como:

```
Definition extract (t: Λα): Λα :=
   let t' := t.car in
   match t with
   | Lam x b => {| car := b; eqv: t.eqv; prf := _ |}
   | _ => t.
```

Observe que o usuário precisa extrair o tipo envolto no *setoid*, já que não é possível realizar um casamento em tipos não-indutivos, e reconstruir o *setoid* no retorno da função. Este padrão de código, que envolve extração e reconstrução, se agrava quando há ocorrência de mais de um *setoid*, conhecido como "*setoid hell*", tornando-se bastante oneroso e complexo, pois introduz mais problemas do que soluções (44).

3.4 Classes de tipos e *setoids*

Afim de mitigar algumas dessas deficiências descritas na seção anterior, Coq define *setoids* como uma classe de tipo:

```
Class Setoid (A: Type) := {
   equiv: relation A ;
   setoid_equiv :> Equivalence equiv
}.
```

Dessa forma, o usuário não precisa lidar diretamente com a relação e sua prova de equivalência, pois o Coq aplica busca de provas afim de encontrar uma instância `Setoid` adequada. Para resolver o problema da reescrita, introduzida na seção 3.3, Coq possui uma implementação de reescrita generalizada (45), também conhecida como reescrita *setoids*, que permite substituir termos equivalentes, isto é *setoids*, através de uma interface simplificada. Para tanto, o usuário precisa informar sobre quais funções é seguro reescrever, em outras palavras, se a função é respeitosa, ou *respectful*:

Definição 9 (Função Respeitosa)**.** *Sejam X e Y tipos, com suas respectivas relações de equivalências $\approx_X$ e $\approx_Y$. Uma função $f : X \to Y$ é respeitosa, se esta preserva as relações de equivalência para todas as entradas:*

$$\forall a\ b \in X,\ a \approx_X b \to f(a) \approx_Y f(b)$$

A próxima seção apresenta um conjunto de exemplos mais aprofundados, englobando classes de tipos, *setoids* e reescrita generalizada.

3.5 Exemplo de classes de tipo e reescrita generalizada

Com o intuito de exemplificar tudo que foi apresentado, abaixo apresento uma implementação da classe de tipos grupo:

```
1  Class Neutral A := neutral: A.        Notation ε := neutral.
2  Class Operator A := op: A → A → A.    Infix "+" := op.
3  Class Inverse A := inv: A → A.        Notation "- x" := (inv x).
4
5  Class Group (A : Type)
6    `{Ntr: Neutral A, Opr: Operator A, Inv: Inverse A, Equiv A} : Prop := {
7          grp_setoid :> Equivalence (≡@{A});
8          grp_op_proper :> Proper ((≡@{A}) ⇒ (≡@{A}) ⇒ (≡@{A})) (+);
9          grp_inv_proper :> Proper ((≡@{A}) ⇒ (≡@{A})) (-);
10         (* ... *)
11         grp_left_inv : ∀ (x : A), (-x) + x ≡@{A} ε@{A};
12 }.
```

Linhas 1–3 são classes operacionais. O objetivo é dar um nome canônico, e notação, as operações e elementos de grupos. Temos o elemento neutro (`neutral ε`), a operação binária (`op +`) e função inversa (`inv -`). Entre as linhas 5–12 tem-se a implementação da classe predicada `Group` (a maior parte é omitida para simplificação da discussão), seu objetivo é agrupar os axiomas de grupo. A classe recebe cinco parâmetros, um explícito tipo `A`, seguindo de quatro implícitos generalizados (entre `` `{} ``): três instâncias das classes operacionais e por último uma instância da classe `Equiv`. Esta última permite definir grupo para um *setoid* de `A`, separando em classe operacional, referenciada pelo nome `equiv` com notação `≡`, e classe predicada (`Equiv`) contendo a relação de equivalência. Os parâmetros entre `` `{} ``, como dito, são implicitamente generalizados, isto é, caso dependam de outros parâmetros implícitos, o Coq os generaliza e incluem na lista de parâmetros implícitos, simplificando o trabalho do usuário, pois este não precisa memorizar todos os parâmetros necessários a uma classe. Concluindo a parte sintática, as notações terminadas em `@{A}` fornecem explicitamente parâmetros, que diferentemente seriam inferidos implicitamente. As linhas 7–11 são as propriedades da classe, onde apenas a 11 é referente aos axiomas de grupo (as demais foram omitidas). As propriedades entre 7–9 são necessárias a reescrita *setoid*: `grp_setoid` é uma prova de equivalência para a relação `≡@{A}` (classe `Equivalence`) e `grp_op_proper` e `grp_inv_proper` são provas de que a operação binária e inversão são próprias, isto é, contextos nos quais é seguro realizar reescrita generalizada. `Proper` espera dois argumentos: uma assinatura e uma função. A assinatura descreve as relações de equivalência para as entradas e saída da função, isto é, `Proper ((≡@{A}) ⇒ (≡@{B}) ⇒ (≡@{C})) g` é equivalente a:

$$\forall (xy : A)(zw : B),\ x \equiv_A y \to z \equiv_B w \to g(x, z) \equiv_C g(y, w)$$

Assim, o assistente sabe como reescrever `x + y` para `z + y`, dado uma prova de `x ≡ z`.

O mecanismo de reescrita generalizada simplifica bastante as provas, eliminando quase completamente as inconveniências envolvendo *setoid*. Abaixo apresento um breve resultado utilizando as técnicas descritas acima, comparando uma propriedade de teoria de grupos e seu equivalente no assistente:

Lema 1. *Seja $x \in G$, tal que G é um grupo. A função inversa é involutiva:*

$$
\begin{aligned}
& x = \\
\text{id. esquerda} \quad & \varepsilon + x = \\
\text{inv. esquerda} \quad & (-(-x) + -x) + x = \\
\text{associatividade} \quad & -(-x) + (-x + x) = \\
\text{inv. esquerda} \quad & -(-x) + \varepsilon = \\
\text{id. direita} \quad & -(-x)
\end{aligned}
$$

```
Lemma grp_inv_involutive {G: Type}
  (x: G) `{Group G}: -(-x) ≡@{G} x.
Proof with auto.

rewrite <-(grp_left_id x) at 2;
rewrite <-grp_left_inv;
rewrite <-grp_assoc;
rewrite grp_left_inv;
rewrite grp_right_id...

Qed.
```

À esquerda tem-se a prova informal, enquanto à direita sua prova formal. O lema `grp_inv_involutive` tem como parâmetros: um tipo `G` (implícito), um termo `x` de `G` e uma prova implícita de que `G` possui uma instância de grupo. Por estar implicitamente generalizada (entre `` `{} ``), o Coq inclui, (implicitamente) todos os parâmetros de `Group`. Portanto, temos acesso a um operador binário, uma função inversa, um elemento neutro e uma equivalência para `G`, além das propriedades de grupo e reescrita nos operadores `+` e `-`. Sem o mecanismo de reescrita generalizada fornecido pelo Coq, a demonstração do lema seria, em sua maioria, aplicação de propriedades de infraestrutura, em outras palavras, permite uma formalização sucinta, próximo a prova informal, enquanto os detalhes ficam a cargo do assistente de provas.

4 Formalizações de conjuntos nominais

Neste capítulo apresentamos duas metodologias distintas de implementação da teoria nominal, desenvolvidas no assistente de provas Coq versão 8.13.2. A primeira metodologia na Seção 4.2, proposta por Copello *et. al.* (13), está relacionado a uma classe de implementações pragmáticas, também denominadas de "técnicas nominais" (11, 13, 46, 14, 7). A segunda na Seção 4.3, desenvolvida por Choudhury em sua dissertação de mestrado (15), apresenta uma formalização de conjuntos nominais propriamente dito.

A técnica nominal de Copello *et al.*, tem como principal característica: considerar a permutação de nomes como uma operação básica, por exemplo ao definir substituição de termos ou α-equivalência, pois esta apresenta um melhor comportamento devido a impossibilidade da captura de variável livre, simplificando a verificação. Outro ponto importante em (13) é a derivação de um princípio de indução/recursão α-estrutural, simulando efetivamente a condição de variável de Barendregt (47, BVC). A aplicação da teoria nominal em trabalhos de verificação formal torna-se complexa sem uma infraestrutura automatizada (6). Isto se deve a sua complexidade de implementação, em relação a outros métodos como índices de de Bruijn (1) e *Locally Nameless* (2). Portanto, propomos uma investigação da formalização de conjuntos nominais em Coq, baseado na abordagem de Choudhury (15), como uma biblioteca de suporte para projetos futuros.

Começamos o capítulo com a discussão de nomes e permutações, visto que a implementação dos mesmos é idêntica em ambas as metodologias.

4.1 Átomos e Permutação

Definimos nomes como estruturas atômicas, ou seja, indivisíveis. A estrutura interna dos átomos é irrelevante, pois estamos interessados apenas em seus identificadores, assim como ponteiros em algumas linguagens de programação. No entanto, certas condições precisam ser satisfeitas: o conjunto de nomes deve ser contável infinito e com igualdade decidível. Dessa forma, é possível obter um nome novo, também chamado de fresco, e decidir quando dois nomes são diferentes ou iguais, sob quaisquer circunstâncias.

Módulos e assinaturas do Coq permitem definir estruturas de dados abstratas, ideal para implementação de átomos. Enquanto módulos são semelhantes a outras implementações em linguagens de programação convencionais, no sentindo de agrupar definições sobre um único nome, assinaturas tem origem na linguagem OCaML (48), com influências de linguagens do paradigma orientado a objetos, na qual permitem: ocultar, limitar o acesso e visibilidade do conteúdo externo ao módulo. Abaixo segue a implementação de átomos

utilizando módulos:

```
Module Type ATOMIC.
  Parameter t: Set.
  Axiom cnt: Countable t.
  Axiom dec: EqDecision t.
  Axiom inf: Infinite t.
End ATOMIC.
Notation name := Atom.t
```

```
Module Atom : ATOMIC.
  Definition t := nat.
  Instance cnt: Countable t := nat_countable.
  Instance dec: EqDecision t := nat_eq_dec.
  Instance inf: Infinite t := nat_infinite.
End Atom.
```

O módulo `Atom` agrupa a definição de átomo `t`, juntamente com suas propriedades, via classes de tipos: `cnt`, `dec` e `inf`, respectivamente provas da contabilidade, decidibilidade e infinitude. Átomos são implementados como números naturais, pois estes possuem as propriedades esperadas. Afim de ocultar a implementação, aplica-se a assinatura `ATOMIC` ao módulo `Atom`, definindo-o como um conjunto qualquer, pertencente ao universo **`Set`**. Ao longo da formalização utilizamos a palavra-chave `name` como apelido para `Atom.t`.

A definição da função de transposição (Definição 1) é direta, onde recebe um par de nomes `'(a,b)`, um nome `c` e retorna um nome:

```
Definition swap '(a,b) c: name :=
   if decide (a = c) then b else (if decide (b = c) then a else c).
```

O operador `decide` é proveniente da classe `EqDecision`, que recebe uma relação e retorna o procedimento de decisão para esta relação. Dado que permutações podem ser decompostas em transposições, a representamos como uma lista de pares de nomes:

```
Notation perm := (list (name * name)).
Notation "( a , b )" := (@cons (name * name) (a,b) (@nil _)).
Notation "( )" := (@nil (name * name)).
```

Como essa representação não é única, por exemplo `(a,a)` e `()` são representações do elemento neutro, apresentamos uma relação de equivalência para permutações para identificar aquelas que produzem o mesmo efeito, introduzindo nosso primeiro *setoid*:

```
Definition swap_perm (p: perm) (a: name): name :=
   foldl (λ x y, swap y x) a p.
Instance perm_equiv: Equiv perm :=
   λ (p q: perm), ∀ (a: name), swap_perm p a = swap_perm q a.
Instance perm_equivalence: Equivalence perm_equiv. Proof. (* ... *) Qed.
```

Em que, `swap_perm` aplica todas as transposições de uma lista em um nome. Outra vantagem desta representação é podermos definir facilmente a permutação inversa, revertendo a lista. Caso tivéssemos representado permutações como funções bijetoras, não seriamos capazes de obter a função inversa para qualquer permutação. Mesmo com a prova da existência de tal inversa, sua extração só seria possível com adição do axioma da escolha única, que não é válido no cálculo de construções (49).

4.2 Copello *et al.*

O ponto central dos trabalhos de Copello *et al.* (13, 14) é a derivação de um princípio de indução/recursão α-estrutural para o cálculo λ em uma teoria de tipos construtiva (Agda). Os únicos requisitos para isso são: a existência da ação de permutação, sobre os termos do cálculo, e a α-equivalência definida via permutação de nomes. Diferentemente do pacote nominal do Isabelle/HOL, que define uma estrutura de dados para representar a classe de α-equivalência, Copello *et al.* trabalha com termos λ puros (2.4), o que ele chama de *raw terms*. Para tanto, introduzem o conceito de α-compatibilidade:

Definição 10 (α-Compatibilidade)**.** *Um predicado $P : \Lambda \to \mathbb{P}$, sobre os termos do cálculo λ é dito α-compatível quando $M \sim_\alpha N$ implica $PM \leftrightarrow PN$. Seja X um conjunto, uma função $f : \Lambda \to X$ sobre os termos do cálculo λ é dito fortemente α-compatível quando se $M \sim_\alpha N$ então $f(M) = f(N)$.*

Dessa forma, ao aplicar uma propriedade ou função da classe, escolhe-se um representante apropriado da α-classe de equivalência, na qual todos os nomes ligados são diferentes dos nomes livres, via permutação de nomes, efetivamente simulando a BVC.

Com isso, é possível derivar um princípio de indução (`term_aeq_rect`) α-estrutural para propriedades α-compatíveis (`αCompat`):

```
1 Definition term_aeq_rect:
2   ∀ P: Λ → Type, αCompat P →
3     (∀ a, P 'a) →
4     (∀ m n, P m → P n → P (m × n)) →
5     {L & ∀ m a, a ∉ L → P m → P (\a · m)} →
6     ∀ m, P m.
7 Proof. (* ... *) Qed.
```

Para simplificação, omitimos a definição dos termos do cálculo λ (`Λ`), representada pelas notações: `'a` para variável, `m × n` aplicação e `\a · m` abstração. A definição `term_aeq_rect` é bem similar ao princípio de indução/recursão estrutural dos termos do cálculo λ, com exceção do caso da abstração (linha 5), na qual introduze-se um tipo Σ com notação `{a: A & T a}`. `term_aeq_rect` pode ser utilizado tanto como princípio de indução quanto um combinador de recursão. No caso da indução, sabemos que existe `L` tal que a variável ligada é fresca. Já para o combinador de recursão, o tipo Σ permite o usuário fornecer um conjunto de variáveis a serem evitados, isto é, a cada chamada recursiva, o combinador troca as variáveis ligadas por outras não contidas em `L`. Afim de facilitar o uso de `term_aeq_rect` como combinador de recursão, definimos o operador `LIt`. Para utiliza-lo é necessário especificar as funções para cada construtor do cálculo λ: `fvar`, `fapp` e `fabs` e um conjunto `L` de nomes a serem evitados.

```
Definition LIt {A: Type} (L: nameset) (l: Λ)
           (fvar: name → A) (fapp: A → A → A) (fabs: atom → A → A) : A :=
```

```
  term_aeq_rect (λ _, A) (λ _ _ _, id) hv (λ _ _, hp)
                (existT _ L (λ _ b _ r, fabs b r)) l.
```

Conseguimos provar, também, que `LIt` sempre produz funções fortemente α-compatíveis (`LItStrongCompat`).

```
Lemma LItStrongCompat {A} fvar fapp fabs L:
  αStrongCompat (@LIt A fvar fapp fabs L).
```

Como exemplo, podemos definir a substituição de termos do cálculo λ da seguinte maneira:

```
Definition subst_term (M N: Λ) (a: name): Λ :=
  let L := ({a} ∪ FV(N) ∪ FV(M)) in
  @LIt Λ (λ b, if decide (a = b) then N else 'b) App Abs L M.
Notation " '[' a ':=' N ']' M " := (subst_term M N a)
```

No caso da abstração não é necessário verificar a captura de variável livre, pois temos a garantia de que as variáveis ligadas serão sempre diferentes do conjunto `L`, dessa forma apenas propagamos a chamada recursiva para o corpo da abstração. A título de exemplo, supondo a aplicação da substituição a uma abstração: $[b := (ac)](\lambda a.ab)$. Note que, com uma implementação errônea da substituição poderia ocorrer a captura da variável a no termo a ser substituído: $\lambda a.a(ac)$. Entretanto, a definição acima, através do combinador `LIt`, a substituição expande para $\lambda(\langle a\ d\rangle a).((\langle a\ d\rangle(ab))[b := (ac)])$, com $d \notin L$. Aplicando as transposições, tem-se: $\lambda d.((db)[b := (ac)]) = \lambda d.d(ac)$. Por estarmos trocando todas as variáveis ligadas por variáveis frescas, temos a segurança de que não haverá captura de variável pela substituição.

Infelizmente, o método possui algumas demonstrações longas e complexas, com aplicações não triviais de transitividade, o que dificulta bastante a automação da técnica. Dessa forma, sua aplicação em formalizações com múltiplas gramáticas é dificultada, pois a maioria das definições, como `term_aeq_rect`, precisam ser completamente redefinidas manualmente para cada gramática (sintaxe).

4.3 Choudhury

Uma solução para os problemas da abordagem em (13), como a repetição de código, seria o desenvolvimento de uma biblioteca de conjuntos nominais, possibilitando a automatização da metodologia.

Na busca de desenvolvimentos, e bibliotecas, de conjuntos nominais em assistentes de provas, pouco foi encontrado, com exceção dos trabalhos produzidor por Urban e seu grupo de métodos nominais[1] (12, 50, 51, 52, 53), e a dissertação de mestrado de Choudhury(15). Entretanto, podemos qualificar os trabalhos do Urban como formalizações de **técnicas** nominais, pois estão interessados apenas na aplicação das ideias centrais da

[1] *Nominal Methods Group* <https://nms.kcl.ac.uk/christian.urban/Nominal/>

teoria nominal em assistentes de provas, assim como Copello. O que diferencia ambos os projetos é a aptidão de mecanização e automação do pacote nominal de Urban. Dessa busca, o único trabalho que realmente propõe formalizar a noção de conjuntos nominais é (15), com uma representação **construtiva** no assistente de provas Agda (54).

Destacamos construtivo porque, a definição (clássica) de menor suporte, apresentado no Capítulo 2, nem sempre é computável construtivamente (55, 56). Apesar de também não ser sempre computável para assistentes baseados em lógica clássica, dependentes do Axioma da Escolha (9), como o Isabelle/HOL. Uma solução proposta por (12), igualmente abraçada por Choudhury, é definir um conjunto com uma "noção de permutação" e posteriormente apresentar **algum** suporte S, tal que $\text{supp} \subseteq S$. Nos casos em que não é possível computar o menor suporte automaticamente, Urban utiliza heurísticas ou o usuário é responsável por fornecer um suporte. Entretanto, em lógicas construtivas, o problema vai além da simples possibilidade de não existência. Mesmo que postulado, o menor suporte leva a derivação do Princípio Limitado Fraco de Onisciência (56), um princípio não construtivo. Portanto, seu uso deve ser evitado em desenvolvimentos construtivos.

Nosso desenvolvimento segue basicamente o trabalho de Choudhury, com algumas exceções. A principal é a fundamentação utilizada para caracterização de conjuntos nominais. Choudhury opta por utilizar teoria de categorias, entretanto a falta de uma biblioteca de categorias em Coq, robusta e livre de axiomas, impossibilitou seguir por este caminho. Por exemplo, a única teoria livre de axiomas (57), desenvolvida para versão 8.8.2 do Coq, possui muitos problemas de inconsistência de universos na versão utilizada neste trabalho (8.13.2), possivelmente relacionado a formalização de categorias grandes (58). Outras diferenças propostas à representação de Choudhury, menos relevantes, serão introduzidos ao longo da seção.

4.3.1 Ação de permutação

Afim de garantir que a definição de permutações (seção 4.1) realmente implementa permutações, definimos uma instância de `Group` (seção 3.4) para `perm`, com a concatenação como operador binário, reversão de lista como função inversa e lista vazia como elemento neutro:

```
Instance perm_neutral: Neutral perm := @nil (name * name).
Instance perm_operator: Operator perm := @app (name * name).
Instance perm_inverse: Inverse perm := @reverse (name * name).
Instance PermGrp: Group perm. Proof. (* ... *) Qed.
```

Definimos a classe ação de permutação parametrizada por uma instância de grupo:

```
1 Class Action A X := action: A → X → X.    Infix "•" := action.
2 Class GAction `(Group G) (X: Type) `{Act: Action G X, Equiv X}: Prop := {
3   gact_setoid :> Equivalence(≡@{X});
```

```
4     gact_proper :> Proper ((≡@{G}) ⇒ (≡@{X}) ⇒ (≡@{X})) (•);
5     gact_id: ∀ (x: X), ε@{G} • x ≡@{X} x;
6     gact_compat: ∀ (p q: G) (x: X), p • (q • x) ≡@{X} (q + p) • x
7  }.
8
9  Class PermAct X := prmact :> Action perm X.
10 Class Perm (X : Type) `{P : PermAct X, Equiv X} :=
11    prmtype :> GAction PermGrp X (Act := @prmact X P).
```

Na linha 2, grupo é um parâmetro explícito generalizado `(Group G), ou seja, o parâmetro para a instância de grupo é explícita, enquanto o resto dos parâmetros (implícitos) de grupo são introduzidos implicitamente. Fazemos isso, pois queremos explicitar a parametrização de ação de grupo em relação a uma instância de grupo. Por exemplo, nas linhas 10–11 definimos tipos de permutação (Perm) parametrizado por uma instância específica de grupo de permutação (PermGrp). As linhas 3 e 4 permitem reescritas no contexto da operação de ação, respeitando as equivalências de grupo (≡@{G}) e do tipo que sofre a ação (≡@{X}). Enquanto as linhas 5 e 6 são as propriedades usuais de ação à esquerda de grupo. A classe Perm é justamente a definição de tipos de permutação proposta em (12).

4.3.2 Conjuntos nominal

Conjuntos nominais são conjuntos de permutação na qual todos os membros são suportados (Definição 8). Por isso, a classe Nominal definida abaixo, é uma extensão direta de Perm, indicados pela propriedade nperm:

```
Context (X: Type) `{Perm X}.
Class Support A := support: A → nameset.
Class Nominal `{Support X}: Prop := {
   nperm :> Perm X;
   support_spec: ∀ (x: X) (a b: name),
      a ∉ (support x) → b ∉ (support x) → ⟨a,b⟩ • x ≡@{X} x
}.
```

Diferentemente de Group e GAction, a classe Nominal é definida utilizando o mecanismo de seção e contexto do Coq. Através do vernacular **Context**, Coq introduz (explicitamente) o tipo X e uma prova (implícita) de que é um conjunto de permutação. Isso nos dá acesso a ação e a relação de equivalência, definida por Perm X, que é posteriormente utilizada em support_spec. Ao encerrar a seção, o assistente inclui a lista de argumentos implícitos da classe Nominal, todos os parâmetros introduzidos no contexto. Outra alternativa de especificação seria: **Class** Nominal (X: **Type**) `{Support X, Perm X}: **Prop** := {(* ... *)}, entretanto estaríamos introduzindo uma redundância de Perm X desnecessária, além de não ter sentido semântico, pois Nominal já possui uma prova de Perm X pelo método nperm, apesar de não afetar provas posteriores. O problema semântico é semelhante a questão de herança e composição de classes em linguagens orientadas a objetos. Perm

como parâmetro designaria uma "composição" de classes, enquanto como membro de classe indicaria uma relação de herança.

4.3.2.1 Instâncias de conjuntos nominais e permutação

Afim de apresentar uma instância `Nominal` para um tipo qualquer `X`, seguimos os seguintes passos: (1) definir uma relação de equivalência para `X`, ou seja, mostrar instâncias `eqX: Equiv X` e `Equivalence eqX`; (2) estipular uma ação de permutação `PermAct X` com uma prova de que respeita as propriedades da mesma `Perm X`; (3) definir suporte (`Support X`) e finalmente mostrar que todo elemento é suportado (`Nominal X`). A seguir apresentamos algumas instâncias interessantes de `Nominal` e `Perm` (12).

Bool é trivialmente nominal, com igualdade sintática como equivalência, ação nula e suporte vazio:

```
Instance bool_equiv: Equiv bool := λ b1 b2, b1 = b2.
Instance bool_act: Action bool := λ _ b, b.
Instance bool_perm: Perm bool. Proof. (* ... *) Qed.
Instance bool_supp: Support bool := ∅.
Instance bool_nom: Nominal bool. Proof. (* ... *) Qed.
```

Pares (ou tuplas) são representados pela sintaxe: `(a,b)` sendo `a` e `b` termos Gallina. As funções `fst` e `snd` são operadores de projeção, respectivamente para o primeiro e segundo elemento do par. A equivalência para pares é dado quando suas projeções são equivalentes:

```
Instance pair_equiv `{Equiv X, Equiv Y}: Equiv (X * Y) :=
  λ '(x1,y1) '(x2,y2), (x1 ≡@{X} x2) ∧ (y1 ≡@{Y} y2).
```

Ação sobre um par é a propagação da mesma para suas projeções:

```
Instance pair_act `{Action X, Action Y}: Action (X * Y) :=
  λ (p: perm) '(x,y), (p • x, p • y).
```

Tendo definido equivalência e ação, podemos mostrar que par é um tipo de permutação quando suas projeções também são:

```
Instance pair_perm `{Perm X, Perm Y}: Perm (X * Y).
Proof. (* ... *) Qed.
```

Suporte é definido como a união dos suportes das projeções:

```
Instance pair_supp `{Support X, Support Y}: Support (X * Y) :=
  λ '(x,y), (support x) ∪ (support y)
```

E finalmente, assim como `Perm`, mostramos que par é um tipo nominal, desde que seus membros também sejam:

```
Instance pair_nom `{Nominal X, Nominal Y}: Nominal (X * Y).
Proof. (* ... *) Qed.
```

Funções são mais problemática que os exemplos anteriores. Vamos somente demonstrar que funções formam um conjunto de permutação, se seu contradomínio e domínio também forem conjuntos de permutação. Isso introduz duas complicações: primeiro, não podemos utilizar igualdade sintática como relação, pois seria necessário assumir a extensionalidade funcional; segundo, temos que garantir que tais funções respeitem as equivalência de seus domínios e contradomínios. Para contornar este problema, precisamos definir o subconjunto de funções respeitosas. Utilizamos uma solução proposta pelo projeto Iris[2], um *framework* de verificação de programas concorrentes através de lógica de separação (59, 60). Define-se um registro que contem uma função (`f_car`) associada a um certificado de respeitabilidade (`f_proper`):

```
Context (A B: Type) `{Perm A, Perm B}.
Record proper_perm_fun: Type := ProperPermFun {
   f_car :> A → B;
   f_proper: Proper ⇝ ((≡@{A}) ⇒ (≡@{B})) f_car
}.
Notation "A ⇝ B" := (proper_perm_fun A B).
Notation "'λp' x .. y , t" := (* ... *)
```

O símbolo `:>` (em `f_car`) e as notações fornecem uma camada de abstração a definição, permitindo manusear `proper_perm_fun` como uma função padrão do Coq. A partir daqui, temos tudo que precisamos, então seguimos o roteiro descrito no início da seção. Definimos uma relação de equivalência para funções respeitosas:

```
Instance perm_fun_proper_equiv `{Equiv B}: Equiv (A ⇝ B) :=
   λ f g, ∀ (a: A), f a ≡@{B} g a.
```

Seguido da ação sobre funções respeitosas:

```
Instance perm_fun_proper_act `{PermAct A, PermAct B}: PermAct (A ⇝ B) :=
   λ r (f : A ⇝ B), (λp (a: A), r • f(-r • a)).
Proof. (* ... *) Defined.
```

Note que estamos definindo ação sobre funções respeitosas, veja o símbolo `λp`. Note também a necessidade de chamar o ambiente de provas, por meio do vernacular **`Proof. (* ... *) Defined`**, pois precisamos garantir que esta definição de ação seja respeitosa, isto é mostrar uma instância para:

```
Proper (≡@{A} ⇒ ≡@{B}) (λ a: A, r • f (-r • a))
```

que é equivalente demonstrar:

```
∀ (x y: A), x ≡@{A} y → r • f (-r • x) ≡@{B} r • f (-r • y)
```

Como a ação de permutação (`•`) e a função `f: A ⇝ B` são respeitosas, e por hipótese sabemos `x ≡@{A} y` podemos substituir `y` por `x` a direta da relação `≡@{B}`, obtendo:

```
r • f (-r • x) ≡@{B} r • f (-r • x)
```

[2] <https://iris-project.org/>

Por `≡@{B}` ser um relação de equivalência a prova é finalizada trivialmente por reflexividade. Finalizamos com a prova de que o conjunto de funções respeitosas são um conjunto de permutação:

```
Instance perm_fun_proper_perm `{Perm A, Perm B}: Perm (A ⇀ B).
Proof. (* ... *) Qed.
```

Relembrando, para mostrar uma instância de `Perm`, precisamos de: (1) uma prova de equivalência para uma relação de `A ⇀ B`; (2) uma prova de que a ação de permutação é respeitosa para `A ⇀ B`; (3,4) demonstrações das propriedades `gact_id` e `gact_compat` para `A ⇀ B`. A prova para (1) é trivial e omitimos (4) pois aplicam-se o mesmo raciocínio de (3). Destacamos parâmetros implícitos através da notação `@{}` quando necessário.

(2) Respeitabilidade para ação de permutação é equivalente a exibir uma instância `Proper`, isto é:

```
∀ (f g: A ⇀ B) (x y: perm), f ≡ g → x ≡ y →
   x •@{B} f (-x •@{A} a) ≡@{B} y •@{B} g (-y •@{A} a)
```

Por hipótese temos `Perm A` e `Perm B`, portanto sabemos que suas ações (`•@{A}` e `•@{B}`) são respeitosas, dessa forma podemos reescrever a equivalência `x ≡ y`, dentro de `f` e `g`, pois também são funções respeitosas, obtendo:

```
x • f (-x • a) ≡ x • g (-x • a)
```

Finalizamos por reflexividade, reescrevendo a equivalência `x ≡ y`:

```
x • f (-x • a) ≡ x • f (-x • a)
```

(3) Queremos mostrar a nulidade da ação vazia à esquerda em funções respeitosas, ou seja:

```
∀ f: A ⇀ B, ε • f ≡ f
```

Abrindo a relação de equivalência para funções respeitosas temos:

```
∀ (f: A ⇀ B) (a: A), (ε • f) a ≡@{A} f a
```

Aplicando ação sobre função:

```
∀ (f: A ⇀ B) (a: A), ε •@{B} f (-ε •@{A} a) ≡@{A} f a
```

Como `•@{A}` e `f` são respeitosas, realizamos as seguintes reescritas: `∀ (g: G), ε • g ≡ g` e `∀ (g: G), ε ≡ -ε`, respectivamente `gact_id` e equivalência do elemento neutro e sua inversa no grupo. Dessa forma, obtém-se os seguintes passos:

```
∀ (f: A ⇀ B) (a: A), f (-ε • a) ≡ f a
∀ (f: A ⇀ B) (a: A), f (ε • a) ≡ f a
∀ (f: A ⇀ B) (a: A), f a ≡ f a
```

O último passo é resolvido por reflexividade.

5 Perspectivas Futuras

Apresentamos neste projeto uma formalização em progresso de conjuntos nominais no assistente de provas Coq. Obtivemos provas mais enxutas, com algumas sendo uma transcrição passo a passo de sua contrapartida informal, comparado com a formalização de Choudhury (61). Tudo isso graças a uma especificação da teoria nominal através classes de tipos associado ao mecanismo de reescrita generalizada do Coq.

Para obtenção de uma biblioteca automatizada três passos são fundamentais: (1) formalização de funções suportadas; (2) abstração de nomes; (3) derivação do princípio de indução/recursão α-estrutural. Nos próximos seis meses, julho à dezembro de 2021, propomos tratar as formalizações citadas. Com isso nosso trabalho estará no mesmo patamar de (15). A partir de janeiro de 2022, entrando nos últimos meses deste projeto, pretendemos reimplementar as técnicas de (13) utilizando a formalização de conjuntos nominais desenvolvida neste trabalho.

Em trabalhos futuros, seria interessante ir além da formalização e desenvolver um sistema de derivação automático do princípio de indução/recursão α-estrutural. Nos últimos anos tem sido desenvolvidas diversos aparatos de metaprogramação no Coq, como o projeto MetaCoq (62), que utiliza a própria linguagem de especificação Gallina como linguagem de metaprogramação, ou Elpi (63, 64) um dialeto de λProlog (65) incorporado no Coq. Assim, sendo possível implementar uma interface mais amigável para o usuário, onde através da definição indutiva da gramática, o sistema será capaz de gerar toda infraestrutura necessária.

Referências

1 BRUIJN, N. de. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, Elsevier BV, v. 75, n. 5, p. 381–392, 1972. Citado 2 vezes nas páginas 3 e 14.

2 CHARGUÉRAUD, A. The locally nameless representation. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 49, n. 3, p. 363–408, may 2011. Citado 2 vezes nas páginas 3 e 14.

3 GABBAY, M. J.; PITTS, A. M. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, Springer Science and Business Media LLC, v. 13, n. 3-5, p. 341–363, jul 2002. Citado 2 vezes nas páginas 3 e 5.

4 PFENNING, F.; ELLIOT, C. Higher-order abstract syntax. In: *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation - PLDI '88*. [S.l.]: ACM Press, 1988. Citado na página 3.

5 HARPER, R.; HONSELL, F.; PLOTKIN, G. A framework for defining logics. *Journal of the ACM (JACM)*, Association for Computing Machinery (ACM), v. 40, n. 1, p. 143–184, jan 1993. Citado na página 3.

6 AYDEMIR, B. et al. Engineering formal metatheory. In: *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '08*. [S.l.]: ACM Press, 2008. Citado 2 vezes nas páginas 3 e 14.

7 AMBAL, G.; LENGLET, S.; SCHMITT, A. HOπ in Coq. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 65, n. 1, p. 75–124, sep 2020. Citado 2 vezes nas páginas 3 e 14.

8 HIRSCHKOFF, D. A full formalisation of π-calculus theory in the calculus of constructions. In: *Lecture Notes in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 1997. p. 153–169. Citado na página 3.

9 PITTS, A. M. Alpha-structural recursion and induction. *Journal of the ACM*, Association for Computing Machinery (ACM), v. 53, n. 3, p. 459–506, may 2006. Citado 2 vezes nas páginas 3 e 18.

10 PITTS, A. M. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge: Cambridge University Press, 2013. (Cambridge Tracts in Theoretical Computer Science, 57). ISBN 9781139084673. Citado 5 vezes nas páginas 3, 4, 5, 6 e 7.

11 AYDEMIR, B.; BOHANNON, A.; WEIRICH, S. Nominal reasoning techniques in Coq. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 174, n. 5, p. 69–77, jun 2007. Citado 2 vezes nas páginas 3 e 14.

12 URBAN, C. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 40, n. 4, p. 327–356, mar 2008. Citado 6 vezes nas páginas 3, 5, 17, 18, 19 e 20.

13 COPELLO, E. et al. Alpha-structural induction and recursion for the lambda calculus in constructive type theory. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 323, p. 109–124, jul 2016. Citado 5 vezes nas páginas 3, 14, 16, 17 e 23.

14 COPELLO, E.; SZASZ, N.; TASISTRO, Á. Machine-checked proof of the church-rosser theorem for the lambda calculus using the barendregt variable convention in constructive type theory. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 338, p. 79–95, oct 2018. Citado 3 vezes nas páginas 3, 14 e 16.

15 CHOUDHURY, P. *Constructive Representation of Nominal Sets in Agda*. Dissertação (Mestrado) — University of Cambridge, jun. 2015. Citado 7 vezes nas páginas 3, 4, 5, 14, 17, 18 e 23.

16 NOMINAL package Isabelle/HOL. Acessado: 06-06-2021. Disponível em: <https://nms.kcl.ac.uk/christian.urban/Nominal/>. Citado na página 3.

17 NIPKOW, T.; PAULSON, L. C.; WENZEL, M. *Isabelle/HOL: a proof assistant for higher-order logic*. [S.l.]: Springer Berlin Heidelberg, 2002. ISBN 3540433767. Citado na página 3.

18 PITTS, A. Nominal techniques. *ACM SIGLOG News*, Association for Computing Machinery (ACM), v. 3, n. 1, p. 57–72, feb 2016. Citado na página 3.

19 ACETO, L. Interview with Murdoch J. Gabbay and Andrew M. Pitts 2019 Alonzo Church Award recipients. *Bulletin of European Association for Theoretical Computer Science*, v. 128, 2019. Citado na página 4.

20 FORMALIZAçãO Choudhury. Acessado: 02-06-2021. Disponível em: <https://github.com/fasapa/nominal-choudhury>. Citado na página 4.

21 FORMALIZAçãO Copello. Acessado: 02-06-2021. Disponível em: <https://github.com/fasapa/nominal-copello>. Citado na página 4.

22 PITTS, A. M. Nominal logic, a first order theory of names and binding. *Information and Computation*, Elsevier BV, v. 186, n. 2, p. 165–193, nov 2003. Citado na página 5.

23 FRALEIGH, J. B. *A First Course in Abstract Algebra*. 7. ed. [S.l.]: Pearson, 2002. ISBN 0201763907. Citado na página 5.

24 HINDLEY, J. R.; SELDIN, J. P. *Lambda-Calculus and Combinators, an Introduction*. [S.l.]: Cambridge University Press, 2008. Citado na página 6.

25 The Coq Development Team. *The Coq Proof Assistant 8.13*. Zenodo, 2021. Disponível em: <https://doi.org/10.5281/zenodo.4501022>. Citado na página 9.

26 COQUAND, T.; HUET, G. The calculus of constructions. *Information and Computation*, Elsevier BV, v. 76, n. 2-3, p. 95–120, feb 1988. Citado na página 9.

27 COQUAND, T.; PAULIN, C. Inductively defined types. In: *COLOG-88*. [S.l.]: Springer Berlin Heidelberg, 1990. p. 50–66. Citado na página 9.

28 PAULIN-MOHRING, C. Inductive definitions in the system Coq rules and properties. In: *Lecture Notes in Computer Science*. [S.l.]: Springer-Verlag, 1993. p. 328–345. Citado na página 9.

29 SØRENSEN, M. H.; URZYCZYN, P. *Lectures on the Curry-Howard isomorphism*. 1. ed. Amsterdam Boston MA: Elsevier, 2006. v. 149. (Studies in Logic and the Foundations of Mathematics, v. 149). ISBN 9780444520777. Citado na página 9.

30 HALL, C. V. et al. Type classes in haskell. *ACM Transactions on Programming Languages and Systems*, Association for Computing Machinery (ACM), v. 18, n. 2, p. 109–138, mar 1996. Citado na página 9.

31 WADLER, P.; BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*. [S.l.]: ACM Press, 1989. Citado na página 9.

32 SOZEAU, M.; OURY, N. First-class type classes. In: *Theorem Proving in Higher Order Logics*. [S.l.]: Springer Berlin Heidelberg, 2008, (Lecture Notes in Computer Science, 5170). p. 278–293. Citado na página 9.

33 Haskell Data.Ord. Acessado: 02-06-2021. Disponível em: <https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Ord.html>. Citado na página 9.

34 GEUVERS, H. et al. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation*, Elsevier BV, v. 34, n. 4, p. 271–286, oct 2002. Citado 2 vezes nas páginas 9 e 10.

35 CRUZ-FILIPE, L.; GEUVERS, H.; WIEDIJK, F. C-CoRN, the constructive coq repository at nijmegen. In: *Lecture Notes in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 2004. p. 88–103. Citado na página 9.

36 GARILLOT, F. et al. Packaging mathematical structures. In: *Lecture Notes in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 2009. p. 327–342. Citado na página 9.

37 COHEN, C.; SAKAGUCHI, K.; TASSI, E. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). In: ARIOLA, Z. M. (Ed.). *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020. (Leibniz International Proceedings in Informatics (LIPIcs), v. 167), p. 34:1–34:21. Citado na página 9.

38 SPITTERS, B.; WEEGEN, E. van der. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, Cambridge University Press (CUP), v. 21, n. 4, p. 795–825, jul 2011. Citado na página 9.

39 CHLIPALA, A. *Certified programming with dependent types : a pragmatic introduction to the Coq proof assistant*. Cambridge, MA: The MIT Press, 2013. ISBN 9780262026659. Citado na página 10.

40 Coq conversion rules. Acessado: 04-06-2021. Disponível em: <https://coq.inria.fr/refman/language/core/conversion.html>. Citado na página 10.

41 CHICLI, L.; POTTIER, L.; SIMPSON, C. Mathematical quotients and quotient types in Coq. In: *Lecture Notes in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 2003. p. 95–107. Citado na página 10.

42 BARTHE, G.; CAPRETTA, V.; PONS, O. Setoids in type theory. *Journal of Functional Programming*, Cambridge University Press (CUP), v. 13, n. 2, p. 261–293, mar 2003. Citado na página 10.

43 BISHOP, E. *Foundations of Constructive Analysis*. [S.l.]: Ishi Press International, 2012. ISBN 4871877140. Citado na página 10.

44 ALTENKIRCH, T. *From setoid hell to homotopy heaven? The role of extensionality in Type Theory*. 2017. Acessado: 04-06-2021. Disponível em: <https://www.cs.nott.ac.uk/~psztxa/talks/types-17-hell.pdf>. Citado na página 11.

45 SOZEAU, M. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, Journal of Formalized Reasoning, v. 2, n. 1, p. 41–62, 2009. Citado na página 11.

46 TASISTRO, Á.; COPELLO, E.; SZASZ, N. Formalisation in constructive type theory of stoughton's substitution for the lambda calculus. *Electronic Notes in Theoretical Computer Science*, Elsevier BV, v. 312, p. 215–230, apr 2015. Citado na página 14.

47 BARENDREGT, H. *The Lambda Calculus. Its Syntax and Semantics*. [S.l.]: College Publications, 2012. (Studies in Logic: Mathematical Logic and Foundations, 40). ISBN 184890066X. Citado na página 14.

48 OCaML Programming Language. Acessado: 7-06-2021. Disponível em: <https://ocaml.org/>. Citado na página 14.

49 MAIETTI, M. E. On choice rules in dependent type theory. In: *Lecture Notes in Computer Science*. [S.l.]: Springer International Publishing, 2017. p. 12–23. Citado na página 15.

50 URBAN, C.; NORRISH, M. A formal treatment of the barendregt variable convention in rule inductions. In: *Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding - MERLIN '05*. [S.l.]: ACM Press, 2005. Citado na página 17.

51 URBAN, C.; BERGHOFER, S. A recursion combinator for nominal datatypes implemented in isabelle/HOL. In: *Automated Reasoning*. [S.l.]: Springer Berlin Heidelberg, 2006. p. 498–512. Citado na página 17.

52 HUFFMAN, B.; URBAN, C. A new foundation for nominal isabelle. In: *Interactive Theorem Proving*. [S.l.]: Springer Berlin Heidelberg, 2010. p. 35–50. Citado na página 17.

53 URBAN, C.; KALISZYK, C. General bindings and alpha-equivalence in nominal isabelle. In: *Programming Languages and Systems*. [S.l.]: Springer Berlin Heidelberg, 2011. p. 480–500. Citado na página 17.

54 BOVE, A.; DYBJER, P.; NORELL, U. A brief overview of Agda - A functional language with dependent types. In: *Lecture Notes in Computer Science*. [S.l.]: Springer Berlin Heidelberg, 2009. p. 73–78. Citado na página 18.

55 SWAN, A. An algebraic weak factorisation system on 01-substitution sets: a constructive proof. *Journal of Logic and Analysis*, Journal of Logic and Analysis, 2016. Citado na página 18.

56 SWAN, A. Some brouwerian counterexamples regarding nominal sets in constructive set theory. *arXiv: 1702.01556*, fev. 2017. <https://arxiv.org/abs/1702.01556> Acessado 30-05-2021. Citado na página 18.

57 WIEGLEY, J. *Category Theory in Coq: An axiom-free formalization of category theory.* Acessado 03-06-2021. Disponível em: <https://github.com/jwiegley/category-theory>. Citado na página 18.

58 GROSS, J.; CHLIPALA, A.; SPIVAK, D. I. Experience implementing a performant category-theory library in Coq. In: *Interactive Theorem Proving*. [S.l.]: Springer International Publishing, 2014. p. 275–291. Citado na página 18.

59 JUNG, R. et al. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices*, Association for Computing Machinery (ACM), v. 50, n. 1, p. 637–650, may 2015. Citado na página 21.

60 JUNG, R. et al. Higher-order ghost state. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. [S.l.]: ACM, 2016. Citado na página 21.

61 CHOUDHURY, P. *Códifo fonte: Constructive Representation of Nominal Sets in Agda.* Acessado: 06-06-2021. Disponível em: <https://www.cl.cam.ac.uk/~amp12/agda/choudhury/html/README.html>. Citado na página 23.

62 SOZEAU, M. et al. The MetaCoq project. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 64, n. 5, p. 947–999, feb 2020. Citado na página 23.

63 DUNCHEV, C. et al. ELPI: Fast, embeddable, λProlog interpreter. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. [S.l.]: Springer Berlin Heidelberg, 2015. p. 460–468. Citado na página 23.

64 TASSI, E. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). Working paper or preprint. 2018. Disponível em: <https://hal.inria.fr/hal-01637063>. Citado na página 23.

65 MILLER, D.; NADATHUR, G. *Programming with Higher-Order Logic*. [S.l.]: Cambridge University Press, 2009. Citado na página 23.