

Fabício Sanches Paranhos

Uma Formalização da Teoria Nominal em Coq

Goiânia, Goiás

2021

Resumo

Segundo a ??, 3.1-3.2, o resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. A ordem e a extensão destes itens dependem do tipo de resumo (informativo ou indicativo) e do tratamento que cada item recebe no documento original. O resumo deve ser precedido da referência do documento, com exceção do resumo inserido no próprio documento. (...) As palavras-chave devem figurar logo abaixo do resumo, antecedidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

Palavras-chave: latex. abntex. editoração de texto.

Sumário

1	INTRODUÇÃO	5
1.1	Estrutura	5
2	REFERENCIAL TEÓRICO	7
3	METODOLOGIA	11
3.1	Coq	11
3.2	Classes de tipos	11
3.3	Igualdade e <i>setoids</i>	12
3.4	Classes de tipos e <i>setoids</i>	13
3.5	Exemplo de classes de tipo e reescrita generalizada	14
4	FORMALIZAÇÃO	17
4.1	Átomos e Permutação	17
4.2	Copello <i>et. al.</i>	18
4.3	Choudhury	20
4.3.1	Ação de permutação	21
4.3.2	Conjuntos nominal	22
4.3.2.1	Instâncias de conjuntos nominais e permutação	22
5	PERSPECTIVAS FUTURAS	25
	REFERÊNCIAS	27

1 Introdução

Ola Pits ([1](#))

1.1 Estrutura

Falar sobre a estrutura do documento.

2 Referencial Teórico

Referencial teórico sobre conjuntos nominais. falar muito breve sobre nomes e simetria/permutação.

Nominal sets capture the notion of name (in)dependence through a simple, and uniform, metatheory based on name permutations. By enriching a structure with (group) action of permutations is possible to define such notion of dependence, where the set of names a structure depends on is called support while its complement is formed by the so-called fresh names. In short, nominal sets gives us the tools to express name dependence for any element of it, only by the means of name permutation, i.e.: if any permutation changes a name from the support of an element, then it'll change such element. (ainda precisa de revisão geral no parágrafo)

Nomes e permutação de nomes são os pilares da teoria nominal (1, 2, 3). Nomes são úteis por seus identificadores únicos, assim como ponteiros em C/C++ e referências em Java ou C#. Sua estrutura interna é abstrata, irrelevante e atômica, isto é, indivisível. Assim, assume-se um conjunto contável¹ infinito de nomes (átomos), com igualdade decidível, denominado \mathbb{A} , representado por pelas letras minúsculas no início do alfabeto: a, b , etc. Transposição de nomes (definição 1), informalmente é uma operação de troca de nomes. Superficialmente, é uma operação sem muito o que adicionar, entretanto como será discutido posteriormente, ela é mais simples, pois, ao contrário da substituição de termos, ela não depende da definição de variável livre, ou ligada, e é imune ao problema da captura de variáveis livres (3).

Definição 1 (Transposição de nomes ou *Swap*). *Transposições são permutações 2-ciclos, representando por $\langle a \ b \rangle : \mathbb{A} \rightarrow \mathbb{A}$ e definido como:*

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

Denoto por $Perm\mathbb{A} = (\mathbb{A}, \circ)$, o grupo de permutação finitas sobre \mathbb{A} , formado pela composição de transposições, com a função identidade como elemento neutro. Reservo as letras minúsculas p, q, r e s para designar membros deste grupo. O interessante desta forma de representação, é que permutações podem ser representadas por uma lista de transposições **Isso é interessante para implementação pq não tem que lidar com funções,**

¹ Não é estritamente necessário que o conjunto de nomes seja contável, veja (1, Exercício 6.2, página 109). Como neste trabalho nomes são implementados como números naturais, incluo contável como uma propriedade bônus.

por exemplo: não é sempre possível extrair a função inversa apenas tendo a definição da função. Mas podemos fazer isso com lista, pois basta reverter a lista!:

$$\langle a \ b \rangle \circ \langle c \ d \rangle \circ \langle e \ f \rangle \Rightarrow \langle a \ b \rangle, \langle c \ d \rangle, \langle e \ f \rangle$$

Permutações agem sobre conjuntos quaisquer por meio da ação de permutação, um caso especial da ação de grupo à esquerda:

Definição 2 (Ação de grupo). *Seja G um grupo, define-se ação à esquerda de G (com elemento neutro representado por ε) sobre um conjunto X qualquer, uma função $(\bullet) : G \times X \rightarrow X$ (é melhor: $(\bullet) : G \rightarrow X \rightarrow X$?), 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). *Conjunto de permutação é aquele dotado de uma ação de permutação. Ação de permutação é uma ação definida pelo grupo $\text{Perm}\mathbb{A}$.*

Exemplos? Ação de permutação nos fornece um mecanismo simples, capaz de expressar independência/dependência de nomes. Mas primeiramente, vamos discutir o significado informal de (in)dependência, no contexto do cálculo- λ , pois seu significado depende da estrutura em questão. O conjunto de nomes um termo do cálculo- λ , módulo α -equivalência, depende, chamado de **suporte**, é justamente o das variáveis livres, pois altera-se seu significado semântico, e sintático, do termo modificando tais nomes. Entretanto, caso não levemos em conta α -equivalência, o grupo de nomes dependentes passa a ser todas as variáveis do termo, livres e ligadas, já que não temos mais a equivalência entre termos com ligantes diferentes. O conjunto de nomes independentes, também chamados de nomes frescos, é o complemento do suporte. Agora formalmente, definimos suporte como:

Definição 4 (Suporte). *Seja X um conjunto de permutação, então $S \subset \mathbb{A}$ suporta $x \in X$ se:*

$$\forall p \in \text{Perm}\mathbb{A}, ((\forall a \in S, p \bullet a = a) \rightarrow p \bullet x = x) \tag{2.4}$$

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, caso a permutação modifique algum elemento de S então ela modificará x . A elegância da definição está no fato de fornecer forma genérica, independente de conjunto, de especificar a dependência de nomes, utilizando apenas ação de permutação de nomes.

Dois pontos extras sobre a definição de suporte: (1) é importante que o suporte seja sempre finito. Se $S = \mathbb{A}$, então não é possível obter novos nomes frescos, afinal, foi

por isso que estipulamos um conjunto infinito para nomes. (2) Podem existir mais de um suporte (S_1 e S_2) para um elemento de um conjunto nominal. Nestes casos é possível mostrar que $S_1 \cap S_2$ também é um suporte (1, Proposição 2.3). O que nos leva a definição de menor suporte e frescor, até agora informal:

Definição 5 (Menor suporte). *Seja X um conjunto de permutação, o menor suporte é dado pela intersecção de todos os suportes de $x \in X$:*

$$\text{supp}(x) \triangleq \bigcap \{S \mid S \text{ suporta } x\} \quad (2.5)$$

Definição 6 (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:*

$$\text{supp}(y) \cap \text{supp}(x) = \emptyset \quad (2.6)$$

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

$$y \# x \leftrightarrow y \notin \text{supp}(x) \quad (2.7)$$

Só falta conjuntos nominais e funções equivariantes. Colocar formalização construtiva aqui? Acho que não

3 Metodologia

3.1 Coq

O assistente de provas Coq (4) 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 (5, 6, 7). Para o desenvolvimento de provas, através do isomorfismo de Curry-Howard (8), 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.

3.2 Classes de tipos

As classes de tipos em Coq, são semelhantes a implementação em Haskell (9), isto é, um mecanismo de polimorfismo *ad-hoc* (10). No assistente de provas Coq, classes de tipos são açúcar sintático (*syntax sugar*) para registros paramétricos dependentes, similar a classes em C++ ou Java. Classes de tipos contam, também, com um sistema de busca de provas e inferência (similar a Prolog). Por consequência, classes de tipos são cidadãos de primeira classe, dessa forma restrições de classe, são apenas parâmetros implícitos (11). Por exemplo, em Haskell, a classe de tipos ordenados depende da classe de tipos com igualdade (12), 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, outro exemplo é a possibilidade de se definir múltiplas instâncias para o mesmo tipo: em Coq, pode-se definir múltiplas instâncias de monoide para os naturais, um para operações de soma outro para multiplicação. Em Haskell, o mesmo efeito só é possível definindo novos tipos (**Sum** e **Prod**) para cada instância de monoide (13).

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 (14, 15, 16, 17). Spitters e van der Weegen (18) 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 predicado. Classes operacionais permitem referenciar operações, como operadores binários em grupos e monoides, concedendo um nome e uma notação canônica. Enquanto classes predicadas agrupam propriedades, no caso de grupos, seus axiomas.

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 parametrizado 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. Igualdade intencional ocorre quando dois termos são convertíveis, isto é, são redutíveis entre si, via regras de conversão¹ do Coq: α , β , ι , δ , ζ e η -expansão. Esta definição é bem útil para reescrita, como pode ser visto no princípio de indução/recursão gerado para **eq**:

eq_rect: \forall (A: **Type**) (x: A) (P: A → **Type**), P x → \forall 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 é possível sem a adição de axiomas extras (19), além de tornar o algoritmo de checagem de tipos indecidível (14). A alternativa é utilizar *setoids*.

Os *setoids*, também conhecidos como conjuntos de Bishop (20, 21), 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

¹ A descrição completa das regras pode ser encontrado no manual oficial: <https://coq.inria.fr/refman/language/core/conversion.html>.

com os detalhes da implementação. Por exemplo, supondo uma representação da classe de α -equivalência para termos- λ , denotado por $\Lambda\alpha$:

Definition $\Lambda\alpha$: `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: $\Lambda\alpha$): $\Lambda\alpha$:=
 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*”, torna-se bastante oneroso e complexo, pois introduz mais problemas do que soluções.

3.4 Classes de tipos e *setoids*

Afim 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 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 (22), 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 7 (Função Respeitosa). *Sejam X e Y tipos, com suas respectivas relações de equivalências \approx_X e \approx_Y . Uma função $f : X \rightarrow Y$ é respeitosa, se esta preserva as relações de equivalência para todas as entradas:*

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

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

MANDAR PARA FORMALIZAÇÃO.

O emprego de *setoids* neste projeto foi essencial para manter a formalização construtiva. A capacidade de representar o conjunto de termos α -equivalentes, *setoids* são essências para manter a formalização construtiva. Além do problema da reescrita, descrito acima, e O axioma da extensionalidade funcional é necessário para demonstrar que o conjunto de funções é um conjunto de permutação (**REFERENCIAL TEÓRICO**). Apesar do sistema formal do Coq ser consistente com o axioma, perde-se a construtividade, o que pode acarretar em atritos futuros, relacionados a extração de código verificado ou o uso da formalização como biblioteca. *Setoids* permitem recuperar a extensionalidade funcional, mantendo a construtividade, através de uma nova relação de equivalência computacional para funções (**VEJA FORMALIZAÇÃO.**).

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: **EXPLICAR EQUIV E SUA RELAÇÃO COM SETOID: SEPARAÇÃO CLASSES OPERACIONAIS E PREDICADAS**

```

1 Class Neutral A := neutral: A.      Notation  $\varepsilon$  := neutral.
2 Class Operator A := op: A  $\rightarrow$  A  $\rightarrow$  A.  Infix "+" := op.
3 Class Inverse A := inv: A  $\rightarrow$  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 ( $\equiv$ @{A});
8     grp_op_proper :=> Proper (( $\equiv$ @{A})  $\Rightarrow$  ( $\equiv$ @{A})  $\Rightarrow$  ( $\equiv$ @{A})) (+);
9     grp_inv_proper :=> Proper (( $\equiv$ @{A})  $\Rightarrow$  ( $\equiv$ @{A})) (-);
10    (* ... *)
11    grp_left_inv :  $\forall$  (x : A), (-x) + x  $\equiv$ @{A}  $\varepsilon$ @{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**, na qual a relação de equivalência é referenciada pelo nome **equiv** e notação \equiv **REMOVER QUADRADO VERMELHO**. Os parâmetros entre `{ }`, como dito, são implicitamente generalizados, isto é, caso dependam de outros parâmetros implícitos, 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, delas 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 $\equiv@{A}$ (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** $((\equiv@{A}) \Rightarrow (\equiv@{B}) \Rightarrow (\equiv@{C}))$ `g` é equivalente a:

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

Assim, Coq sabe como reescrever $x + y$ para $z + y$, dado uma prova de $x \equiv z$.

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:*

Lemma `grp_inv_involutive` $\{G: \text{Type}\}$
 $(x: G) \text{ `}{Group G\}: -(-x) \equiv@{G} x.$
Proof with auto.

	$x =$	
id. esquerda	$\varepsilon + x =$	rewrite <code><- (grp_left_id x) at 2;</code>
inv. esquerda	$-(-x) + -x + x =$	rewrite <code><- grp_left_inv;</code>
associatividade	$-(-x) + (-x + x) =$	rewrite <code><- grp_assoc;</code>
inv. esquerda	$-(-x) + \varepsilon =$	rewrite <code>grp_left_inv;</code>
id. direita	$-(-x)$	rewrite <code>grp_right_id...</code>
		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, o lema estaria tomando pela aplicação de propriedades de infraestrutura, em outras palavras, seu uso possibilita que o usuário concentre-se apenas nas partes importantes do desenvolvimento, enquanto os detalhes ficam sob responsabilidade do assistente de provas.

4 Formalização

Neste capítulo apresento, primeiramente, a implementação de átomos e permutações, pois ambas as formalizações seguintes compartilham da mesma implementação, seção 4.1, junto com duas formalizações, no assistente de provas Coq, baseadas nos trabalhos de Copello *et. al.* (23), seção 4.2, e Choudhury (24), seção 4.3, ambos em Agda. A primeira formalização está relacionado a uma classe de implementações pragmáticas, também denominadas de “técnicas nominais” (23, 25, 26, 27, 28), enquanto a segunda, pertence ao grupo de formalizações de conjuntos nominais, ou sintaxe abstrata nominal (29, 24).

Inicialmente, optei pela técnica nominal adotada por Copello *et. al.*, aonde as principais características são: 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, no trabalho de Copello, é a derivação de um princípio de indução/recursão α -estrutural, simulando efetivamente a BVC. Todavia, como verificado em minhas investigações e outros trabalhos (30), a falta de uma formalização da teoria nominal, ou uma biblioteca mecanizada/automatizada de infraestrutura, acarreta desenvolvimentos desagradáveis, focado em sistemas isolados, pois a adição de outros para análise, por exemplo no estudo de codificações, implicaria na repetição de código. Portanto, alterei o foco do projeto para a investigação da formalização de conjuntos nominais em Coq, apoiado no trabalho de Choudhury, como uma biblioteca de suporte, para projetos futuros.

A formalização se encontra no repositório [bla bla bla](#)

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: conjunto de nomes deve ser infinito, contável e com igualdade decidível. Dessa forma, é possível obter um nome novo (**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 sentido de agrupar definições sobre um único nome, assinaturas tem origem na linguagem OCaml, na qual permitem:

ocultar, limitar o acesso e visibilidade do conteúdo externo ao módulo. Abaixo segue a implementação de átomos através desta técnica:

Module Type ATOMIC.	Module Atom : ATOMIC.
Parameter t : Set.	Definition t := nat.
Axiom cnt : Countable t.	Instance cnt : Countable t := nat_countable.
Axiom dec : EqDecision t.	Instance dec : EqDecision t := nat_eq_dec.
Axiom inf : Infinite t.	Instance inf : Infinite t := nat_infinite.
End ATOMIC.	End Atom.
Notation name := Atom.t	

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 naturais, pois estes possuem as mesmas propriedades esperadas para átomos. 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 utilizo a palavra-chave **name** como apelido para **Atom.t**. **Dizer *oq* Countable, EqDecision e Infinite fazem exatamente?**

A representação de permutações, e função de transposição, não tem segredos, sendo bastante similar a definida no capítulo **REFERENCIAL TEORICO**:

```

Notation perm := (list (name * name)).
Notation "< a , b >" := (@cons (name * name) (a,b) (@nil _)).
Definition swap '(a,b) : name → name :=
  λ c, if decide (a = c) then b else if decide (b = c) then a else c.
Definition swap_perm (p: perm): name → name :=
  λ a, foldl (λ x y, swap y x) a p.
Instance perm_equiv: Equiv perm :=
  λ p q, ∀ a, swap_perm p a = swap_perm q a.
Instance perm_equivalence: Equivalence perm_equiv. Proof. (* ... *) Qed.

```

Além da representação autoexplicativa **perm**, a notação $\langle a, b \rangle$ é útil ao referenciar permutações com um único par de nomes.

Como discutido anteriormente (**QUAL SEÇÃO?**), a representação de permutações não é única. Para isto defino uma relação de equivalência **perm_equiv**, que iguala duas permutações se estas produzem o mesmo resultado sobre todos os nomes, com uma prova (**perm_equivalence**) de que realmente é uma equivalência (reflexividade, simetria e transitividade).

4.2 Copello *et. al.*

O ponto central dos trabalhos de Copello *et. al.* (23, 25) é a derivação de um princípio de indução/recursão α -estrutural para o cálculo- λ em uma teoria de tipos construtiva (Agda), sem a necessidade de uma representação para classe de α -equivalência. Os

únicos requisitos são: a existência da ação de permutação, sobre os termos do cálculo, e α -equivalência definido via permutação de nomes. Copello *et. al.* observa que propriedades/funções, sobre a classe de α -equivalência são invariantes para membros da mesma, denominado α -compatibilidade (definição 8). Dessa forma, ao aplicar uma propriedade, ou função, da classe, pode-se escolher um membro apropriado (via permutação de nomes), ou seja fresco, efetivamente simulando a BVC.

Definição 8 (α -Compatibilidade). *Um predicado P , sobre os termos do cálculo- λ é dito α -compatível se, dados M e N α -equivalentes, implica $PM \leftrightarrow PN$. Uma função F , sobre os termos do cálculo- λ é dito fortemente α -compatível se, dados M e N α -equivalentes, implica $F(M) = F(N)$.*

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

Definition `term_aeq_rect`:

```

 $\forall P: \Lambda \rightarrow \mathbf{Type}, \alpha\text{Compat } P \rightarrow$ 
  ( $\forall a, P \text{ 'a}$ )  $\rightarrow$ 
  ( $\forall m \ n, P \ m \rightarrow P \ n \rightarrow P \ (m \times n)$ )  $\rightarrow$ 
  ( $\{L \ \& \ \forall m \ a, a \notin L \rightarrow P \ m \rightarrow P \ (\backslash a \cdot m)\}$ )  $\rightarrow$ 
   $\forall m, P \ m$ .

```

Proof. (* ... *) **Qed.**

Para simplificação, omito a definição dos termos do cálculo- λ (Λ), representada pelas notações: 'a para variável, $m \times n$ aplicação e $\backslash a \cdot m$ abstração. A definição `term_aeq_rect` é bem similar ao princípio de indução na estrutura dos termos do cálculo- λ , com exceção do caso da abstração. A notação $\{a: A \ \& \ T \ a\}$ introduz um tipo Σ , ou subconjunto, com $T: A \rightarrow \mathbf{Type}$. Sua interpretação depende se estamos utilizando como princípio de indução ou recursão. No caso da indução, temos a garantia de que a variável ligada é suficientemente fresca para um conjunto de nomes L . Já para recursão, ela permite escolher um nome diferente do conjunto, fornecido pelo usuário, em meio a recursão, como pode ser visto abaixo, no operador de recursão `LIIt`, que sempre produz funções fortemente α -compatíveis (`LIItStrongCompat`):

Definition `LIIt {A: Type} (L: nameset) (l: Λ)`

```

  (fvar: name  $\rightarrow$  A) (fapp: A  $\rightarrow$  A  $\rightarrow$  A) (fabs: atom  $\rightarrow$  A  $\rightarrow$  A) : A :=
  term_aeq_rect ( $\lambda \_ , A$ ) ( $\lambda \_ \_ \_ , id$ ) hv ( $\lambda \_ \_ , hp$ )
  (existT  $\_ L$  ( $\lambda \_ b \_ r , fabs \ b \ r$ )) l.

```

Lemma `LIItStrongCompat {A} fvar fapp fabs L:`
 `α StrongCompat (@LIIt A fvar fapp fabs L).`

Para utilizar o princípio é necessário definir as funções para cada construtor do cálculo- λ : `fvar`, `fapp` e `fabs` e um conjunto L de nomes a serem evitados. Por exemplo, a substituição de termos é definida como:

Definition `subst_term (M N: Λ) (a: name): $\Lambda :=$`

```

let L := ({a} u fv N u fv M) in (* fv = support *)
@Lit  $\wedge$  ( $\lambda$  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 , portanto passamos como argumento seu construtor **Abs**, para propagar a recursão ao corpo da abstração.

Infelizmente, a simplicidade aparente do método esconde alguns problemas. Algumas demonstrações tornam-se longas e complexas, com aplicações não triviais de transitividade, o que dificulta bastante a automação da técnica. Isto complica a aplicação em formalizações que contem mais de um sistema, acarretando em repetição de código. Apesar das limitações, o grande triunfo do método é poder utilizar a igualdade padrão do Coq, na permutação e maioria das provas, eliminando a necessidade de *setoids*, simplificando aplicações de reescrita.

4.3 Choudhury

O método de Copello, apesar de simples e direto, introduz repetição de código e complexidade com a introdução de novas estrutura a formalização. Uma possível solução é seria o desenvolvimento de uma biblioteca de conjuntos nominais, encapsulando e abstraindo conceitos, afim reduzir a reutilização de código e mecanizar a implementação de Copello.

Na busca de desenvolvimentos, e bibliotecas, de conjuntos nominais, em assistentes de provas, pouco foi encontrado, com exceção dos trabalhos produzidos por Urban e seu grupo de métodos nominais¹ (31, 32, 33, 34, 35), e a dissertação de mestrado de Choudhury, sob orientação de Pitts (24). Entretanto qualifico os trabalhos do Urban como formalizações das **técnicas** nominais, pois estão interessados apenas na aplicação das ideias centrais da teoria nominal em assistentes, 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 propunha formalizar a noção de conjuntos nominais foi Choudhury, com uma representação **construtiva** no assistente de provas Agda (36).

Meu desenvolvimento segue, basicamente, o trabalho de Choudhury, com algumas exceções. A principal é a fundamentação utilizada para formalizar conjuntos nominais. Choudhury opta por utilizar teoria de categorias, entretanto a falta de uma biblioteca robusta e livre de axiomas de categorias em Coq, impossibilitou seguir por este cami-

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

nho². Outros desvios propostos da representação de Choudhury, menos relevantes, serão introduzidos ao longo do capítulo.

4.3.1 Ação de permutação

Afim de garantir que a definição de permutações (seção 4.1) realmente implementa permutações, estabeleço uma instância de **Group** (seção 3.4) para **perm**, com a concatenação como operador binário, reversão de lista como função reversa 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.
```

Para ação de permutação, desvio um pouco da metodologia de colocar classes operacionais como parâmetros implícitos generalizados na definição de classes predicados (seção 3.2), como pode ser visto na definição de ação 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   prmtime := GAction PermGrp X (Act := @prmact X P).
```

Na linha 2, grupo é um parâmetro explícito generalizado `(Group G)`, ou seja, o parâmetro para uma instância de grupo é explícita, enquanto o resto dos parâmetros de grupo são introduzidos implicitamente. Isto tem um motivo semântico, pois é a de um grupo sobre uma tipo, e prático, em razão de que sempre teremos uma prova de grupo disponível no contexto ao trabalhar com alguma ação. As linhas 3 e 4 são permitem reescritas no contexto da ação, respeitando as equivalências de grupo ($\equiv@{G}$) e do tipo que sofre a ação ($\equiv@{X}$). Enquanto as linhas 5 e 6 são os lemas usuais de ação de grupo a esquerda. Linhas 9–11 especializo ação de grupo para ação de permutação. A classe **Perm** é justamente a definição de tipos de permutação definido por Urban em (31).

² Falar pq? Problemas de inconsistencia de universos causado por categorias grandes?

4.3.2 Conjuntos nominal

Conjuntos nominais são conjuntos de permutação na qual todos os membros são suportados. Por isso, a classe **Nominal** definida abaixo, é uma simples extensão de **Perm**, indicado 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 ao contexto da definição o tipo **X** e uma prova de que é um conjunto de permutação. Isso nos dá acesso a ação e relação de equivalência, definida por **Perm X** e utilizada em **support_spec**. Ao encerrar a seção, o assistente inclui, como argumentos implícitos à classe **Nominal**, todos as propriedades e classes empregues na definição. Outra alternativa de especificação seria: **Class Nominal (X: Type) `{Support X, Perm X}: Prop := {(* ... *)}**, entretanto estaríamos introduzindo uma redundância de **Perm X** desnecessária, e semanticamente não faria sentido, apesar de não afetar provas posteriores. É um fenômeno semelhante a herança e composição em linguagens orientadas a objetos. **Perm** como argumento designaria “composição” de classes, e como membro de classe aponta para herança. Como a semântica correta é no sentido de herança, definimos desta maneira.

4.3.2.1 Instâncias de conjuntos nominais e permutação

Tendo definido a classe nominal, podemos começar a mostrar algumas instâncias triviais da mesma. Para mostrar que 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 apresento algumas instâncias interessantes de conjuntos nominais e conjuntos de permutação.

Bool é trivialmente nominal, com igualdade sintática como equivalência, ação e suporte vazios (o elemento sintático “_” indica, ao Coq, que o parâmetro não será utilizado.):

```
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) com a e b dois tipos quaisquer. As funções `fst` e `snd` extraem, respectivamente, o primeiro e segundo elemento do par. Dois pares são equivalentes, quando seus elementos 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 seus membros:

```

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 conjunto de permutação sabendo que seus membros também são:

```

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

```

Suporte é a união dos suportes dos membros:

```

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 conjunto nominal, desde que seus membros também sejam:

```

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

```

Funções é mais problemática que os exemplos anteriores. Vamos somente demonstrar que funções formam um conjunto de permutação, se sua imagem 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 e segundo, temos que garantir tais funções respeitem as equivalência de seus domínios e imagens. A seguinte definição é errônea pois engloba o conjunto de todas as funções, respeitadas ou não:

```

Instance perm_fun `{Perm X, Perm Y}: Perm (X → Y).

```

Para contornar este problema, precisamos definir o subconjunto de funções respeitadas. Utilizo uma solução proposta na biblioteca Iris, um *framework* de verificação de programas concorrentes através de lógica de separação (37, 38). Definimos um registro que contem uma função (`f_car`) associado a um certificado de respeitosa (`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 " $A \rightsquigarrow B$ " := (proper_perm_fun A B).

Notation "' λp ' $x \dots y$, t " := (* ... *)

O símbolo \rightarrow (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 equivalência computacional para a função carregada `f_car`:

Instance perm_fun_proper_equiv `{Equiv B}: Equiv ($A \rightsquigarrow B$) :=
 $\lambda f g, \forall (a: A), f a \equiv_{@B} g a$.

Seguido da ação sobre funções respeitadas:

Instance perm_fun_proper_act `{PermAct A, PermAct B}: PermAct ($A \rightsquigarrow B$) :=
 $\lambda r (f : A \rightsquigarrow B), (\lambda p (a: A), r \cdot f(-r \cdot a))$.

Proof. (* ... *) **Qed.**

Note que estamos definindo ação sobre funções respeitadas, veja o símbolo λp . Note também a necessidade de chamar o ambiente de provas, por meio do vernacular

Proof. (* ... *) **Qed**, pois precisamos garantir que esta definição de ação seja respeitosa. Finalizamos com a prova de que o conjunto de funções respeitadas são um conjunto de permutação:

Instance perm_fun_proper_perm `{Perm A, Perm B}: Perm ($A \rightsquigarrow B$).

Proof. (* ... *) **Qed.**

5 Perspectivas Futuras

vai ficar faltando suporte de funções, por enquanto só temos conjunto de permutação de funções

Referências

- 1 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 3 vezes nas páginas 5, 7 e 9.
- 2 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 na página 7.
- 3 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 7.
- 4 The Coq Development Team. *The Coq Proof Assistant 8.13*. Zenodo, 2021. Disponível em: <https://doi.org/10.5281/zenodo.4501022>. Citado na página 11.
- 5 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 11.
- 6 COQUAND, T.; PAULIN, C. Inductively defined types. In: *COLOG-88*. [S.l.]: Springer Berlin Heidelberg, 1990. p. 50–66. Citado na página 11.
- 7 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 11.
- 8 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 11.
- 9 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 11.
- 10 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 11.
- 11 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 11.
- 12 Haskell Data.Ord. <https://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Ord.html>. Acessado: 02-06-2021. Citado na página 11.
- 13 Haskell Data.Monoid. <https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Monoid.html#g:3>. Acessado: 28-05-2021. Citado na página 11.
- 14 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 11 e 12.

- 15 CRUZ-FILIPPE, 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 11.
- 16 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 11.
- 17 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. ISBN 978-3-95977-155-9. ISSN 1868-8969. Disponível em: <https://drops.dagstuhl.de/opus/volltexte/2020/12356>. Citado na página 11.
- 18 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 11.
- 19 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 12.
- 20 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 12.
- 21 BISHOP, E. *Foundations of Constructive Analysis*. [S.l.]: Ishi Press International, 2012. ISBN 4871877140. Citado na página 12.
- 22 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 13.
- 23 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 2 vezes nas páginas 17 e 18.
- 24 CHOUDHURY, P. *Constructive Representation of Nominal Sets in Agda*. Dissertação (Mestrado) — University of Cambridge, jun. 2015. Citado 2 vezes nas páginas 17 e 20.
- 25 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 2 vezes nas páginas 17 e 18.
- 26 AMBAL, G.; LENGLET, S.; SCHMITT, A. HO π in Coq. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 65, n. 1, p. 75–124, sep 2020. Citado na página 17.
- 27 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 17.

- 28 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 na página 17.
- 29 AYALA-RINCÓN, M. et al. A formalisation of nominal α -equivalence with a, c, and AC function symbols. *Theoretical Computer Science*, Elsevier BV, v. 781, p. 3–23, aug 2019. Citado na página 17.
- 30 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 na página 17.
- 31 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 2 vezes nas páginas 20 e 21.
- 32 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 20.
- 33 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 20.
- 34 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 20.
- 35 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 20.
- 36 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 20.
- 37 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 23.
- 38 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 23.