

Equipe abnT_EX2

Modelo Canônico de Trabalho Acadêmico com abnT_EX2

Brasil

2018, v<VERSION>

Equipe abnT_EX2

Modelo Canônico de Trabalho Acadêmico com abnT_EX2

Modelo canônico de trabalho monográfico
acadêmico em conformidade com as normas
ABNT apresentado à comunidade de usuários
L^AT_EX.

Universidade do Brasil – UBr
Faculdade de Arquitetura da Informação
Programa de Pós-Graduação

Orientador: Lauro César Araujo
Coorientador: Equipe abnT_EX2

Brasil
2018, v<VERSION>

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, antecidas da expressão Palavras-chave:, separadas entre si por ponto e finalizadas também por ponto.

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

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Sumário

1	INTRODUÇÃO	7
1.1	Estrutura	7
2	REFERENCIAL TEÓRICO	9
3	METODOLOGIA	11
3.1	Coq	11
3.2	Igualdade e <i>setoids</i>	11
3.3	Classes de tipos	12
3.4	Classes de tipos e <i>setoids</i>	13
4	FORMALIZAÇÃO	15
4.1	Átomos e Permutação	15
4.2	Copello <i>et. al.</i>	16
4.3	Choudhury	18
4.3.1	Ação de permutação	19
4.3.1.1	Permutação sobre pares	20
4.3.1.2	Permutação sobre funções	20
4.3.2	Conjuntos nominal	20
5	PERSPECTIVAS FUTURAS	23
	REFERÊNCIAS	25

1 Introdução

Ola Pits ([1](#))

1.1 Estrutura

Falar sobre a estrutura do documento.

Fixpoint

2 Referencial Teórico

3 Metodologia

3.1 Coq

O assistente de provas Coq (2) é sistema gerenciador de desenvolvimentos formais, composto por um motor de provas e três linguagens: Gallina (especificação), Vernacular (comandos) e Ltac (metaprogramação). O vernacular são comandos que permitem interagir com o ambiente de provas, como adicionar definições, realizar consultas e alterar configurações. Ltac é utilizado para metaprogramação, sendo principalmente utilizada no desenvolvimento de táticas, consumidas pelo motor de provas, e automação. Por último, Gallina uma linguagem funcional (especificação) com tipos dependentes, na qual implementa o Cálculo de Construções (Co)Indutivas (3, 4, 5). Para o desenvolvimento de provas, atendendo ao isomorfismo de Curry-Howard (6), Coq usufrui do motor de provas para 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 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 para o tipo `A`. A única forma de construir uma prova `a = b`, é através do construtor `eq_refl`, desde que `a` e `b` sejam iguais sintaticamente, ou convertíveis. 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 → Prop), P x → ∀ y: A, x = y → P y.`

a função `eq_rect` permite reescrever termos, em qualquer contexto `P`, desde que sejam iguais. Em alguns casos, termos sintaticamente diferentes, ou não convertíveis, podem representar semanticamente o mesmo objeto (por exemplo, termos α -equivalentes), mediante uma relação de equivalência. Isto implica na impossibilita a aplicação das táticas padrões de reescrita, para termos equivalentes. Uma solução seria definir conjuntos quocientes, entretanto isto não é possível sem a adição de axiomas (7), pois tornaria o algoritmo de checagem de tipos indecidível (8). A alternativa é utilizar reescrita *setoid*.

Setoids, também conhecidos como conjuntos de Bishop (9, 10), são estruturas formadas por um tipo equipado com uma relação de equivalência, geralmente utilizados

para codificar a “ideia” de conjuntos quocientes. Entretanto introduzem mais problemas do que soluções, um cenário conhecido como “setoid hell”. A implementação de tipos *setoid* não é abstrata, então o usuário, na prática, tem que lidar constantemente com detalhes da implementação (*boilerplate*), como fazer o *lift* do tipo para estrutura *setoid* e gerenciamento manual da aplicação de provas de compatibilidade e lemas customizados de substituição, sem falar quando há mais de um *setoid* envolvido. Para mitigar algumas dessas deficiências, Coq (11) possui uma implementação de reescrita generalizada (também conhecida como reescrita *setoid*), o que permite substituir termos equivalentes através de uma interface limpa e simplificada, em muitos casos transparente. Para tanto, o usuário precisa informar ao sistema quais contextos (em outras palavras: funções), a reescrita é segura, isto é, demonstrando que a função é respeitosa (*respectful*, veja definição 1), por meio de instâncias da classe de tipos **Proper** (VEJA EXEMPLO ABAIXO).

Definição 1 (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 própria, se esta preserva as relações de equivalência, para todas as entradas:*

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

Além do problema da reescrita, descrito acima, e a capacidade de representar o conjunto de termos α -equivalentes, *setoids* são essências para manter a formalização construtiva. 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.).

3.3 Classes de tipos

Não há muito do que dizer sobre classes de tipos em Coq, pois funcionalmente é bem semelhante a implementação em Haskell (12), isto é, um mecanismo de polimorfismo *ad-hoc* (13), com exceção de alguns pontos. Classes de tipos são açúcar sintático (*syntax sugar*) para registros, semelhante a estruturas em C/C++. Entretanto, em Coq, registros são dependentes e paramétricos, ou seja, recebem parâmetros e tipos de membros posteriores podem referenciar membros anteriores. Além do detalhe sintático, classes contam atributos extras: um sistema de busca de provas e inferência (similar a Prolog) e argumentos implícitos. Instâncias de registros são constantes dos mesmos, por consequência, classes de tipos são cidadãos de primeira classe, dessa forma restrições de classe são apenas parâmetros

implícitos (14). Logo, classes de tipos são mais poderosas que sua contrapartida em Haskell, um 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 (15).

Classes de tipos trazem também uma nova possibilidade, mais simples, de construção de uma hierarquia de estruturas algébricas, assim como em Haskell, dentre diversas implementações de sucesso variado (8, 16, 17, 18). Spitters e van der Weegen (19) 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, no caso de grupos, seus axiomas.

3.4 Classes de tipos e *setoids*

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  $\varepsilon$  := 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})) (+);
9     grp_inv_proper :=> Proper ((≡@{A}) ⇒ (≡@{A})) (-);
10    (* ... *)
11    grp_left_inv : ∀ (x : A), (-x) + x ≡@{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 . Os parâmetros entre `{ }`, como dito, são implicitamente generalizados, isto é, caso dependam de outros parâmetros implícitos, Coq os generaliza e incluem a lista de parâmetros implícitos, simplificando o trabalho do usuário, pois este não precisa memorizar

todos os parâmetros necessários a uma classe (**VEJA EXEMPLO ABAIXO**). Concluindo a parte sintática, as notações terminadas em $\text{@}\{A\}$ 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 relação $\equiv\text{@}\{A\}$ (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** $((\equiv\text{@}\{A\}) \Rightarrow (\equiv\text{@}\{B\}) \Rightarrow (\equiv\text{@}\{C\}))$ 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:

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

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

	$x =$	
id. esquerda	$\varepsilon + x =$	rewrite $<-$ (<code>grp_left_id</code> x) at 2;
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 $\{ \}$) Coq inclui, como dito anteriormente, 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 nas nos operadores $+$ e $-$. Como pode ser notado, não há necessidade de lidar diretamente com a implementação de *setoids* e lemas de compatibilidade manualmente. Tornando a formalização quase transparente aos *setoids*.

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.* (20), seção 4.2, e Choudhury (21), seção 4.3, ambos em Agda. A primeira formalização está relacionado a uma classe de implementações pragmáticas, também denominadas de “técnicas nominais” (20, 22, 23, 24, 25), enquanto a segunda, pertence ao grupo de formalizações de conjuntos nominais, ou sintaxe abstrata nominal (26, 21).

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 (27), 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.

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.
  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 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 `{a,b}` 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.* (20, 22) 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 2). 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 2 (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`:

```

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

```

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 `LIt`, que sempre produz funções fortemente α -compatíveis (`LItStrongCompat`):

Definition `LIt {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 `LItStrongCompat {A} fvar fapp fabs L:`

```

   $\alpha\mathbf{StrongCompat} \ (@\mathbf{LIt} \ A \ fvar \ fapp \ fabs \ L).$ 

```

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): Λ :=`

```

  let L := ({a}  $\cup$  fv N  $\cup$  fv M) in (* fv = support *)

```

@LIIt \wedge (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 , 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 (28, 29, 30, 31, 32), e a dissertação de mestrado de Choudhury, sob orientação de Pitts (21). 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 (33). É importante frisar o construtivo, pois na teoria nominal é incompatível em lógicas construtivas, isto se deve ao fato de o menor suporte finito ser não construtivo (34, 35).

Meu desenvolvimento segue, basicamente, o trabalho de Choudhury, com algumas ‘. A principal é a forma, ou caminho, utilizado para formalizar conjuntos nominais. Choudhury opta por utilizar teoria de categorias, entretanto a falta de uma biblioteca robusta e livre

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

de axiomas de categorias em Coq, impossibilitou seguir por este caminho². 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 3.3), 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   prmtype := 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 (28).

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

4.3.1.1 Permutação sobre funções

Ao contrário de Urban, Choudhury não define nenhuma instância de tipos de permutação, portanto sigo a implementação do primeiro aqui para mostrar que existe uma instância de funções para **Perm**. Instancias para nomes, listas e tuplas são trivialmente demonstráveis, entretanto para funções provou-se mais desafiador. Sem *setoids* seria impossível definir tal instância sem adição do axioma da extensionalidade funcional, o que acarretaria numa formalização não construtiva. Entretanto o simples uso de *setoids* também não garante o mesmo.

Assim como feito por Urban, é facilmente demonstrável que, nomes, listas e tuplas (*pares?*) são instâncias de **Perm**.

FUNÇÕES AÇÃO DE PERMUTAÇÃO

4.3.2 Conjuntos nominal

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

Por que Perm é um membro de Nominal? Por que Nominal é um tipo de permutação na qual todo elemento é finitamente suportado.

ao definirmos ação de permutação estou interessado em uma implementação de permutação. Em Haskell isso não seria um problema, pois instâncias são únicas para tipos. Como pode-se haver uma ou mais instâncias de

A repetição de código para diferentes estruturas, devido ao método anterior de Copello, traz uma questão interessante sobre formalizar conjuntos nominais antes de aplicar seu método. Com uma biblioteca já fundamentada sobre conjuntos nominais boa parte seria reaproveitada para desenvolvimentos futuros, quem sabe até mecanizada. Porém a formalização de conjuntos nominais em lógicas construtivas apresentam seus próprios desafios. Aqui apresentamos o desenvolvimento de uma formalização de conjuntos nominais bem próxima a apresentada por Pritam. Com a diferença de aproveitarmos alguns mecanismos presentes no Coq que possibilitam uma formalização mais simples, próxima ao desenvolvimento da metateoria por meios tradicionais. Pritam descreve dois desafios à esta formalização: a não construtividade do menor suporte finito e trabalhar com *setoids*.

Não seria possível desenvolver a teoria, como nos trabalhos de Urban, livre de axiomas. Uma das noções centrais, funções são tipos de permutação, podem ser demonstrados

apenas assumindo a extensionalidade funcional. É importante frisar que extensionalidade não é incompatível com a lógica do Coq, podendo ser incluída como axioma. Entretanto introduzimos um caráter não computacional a formalização, que posteriormente possa impossibilitar extração de código. Ainda, é importante notar que é possível trabalhar com os dois, extensionalidade e extração sem prejudicar, como é feito pela extensão *Program* do Coq, mas isto está fora do escopo deste trabalho. Portanto optamos por trabalhar com *setoids* durante toda a formalização. O que nos diferencia de Pritam é a aplicação de tipos de classes e reescrita generalizada, para alcançar uma formalização mais simples, enxuta, similar as provas de papel e caneta para a metateoria.

Seguimos um modelo proposto por ... para definição das classes de tipos, sendo elas divididas em duas. As classes operacionais servem como referência as operações e notações utilizadas no segundo tipo de classe, fornecendo um nome canônico, uma forma de referenciar as operações. As classes predicados implementam as restrições da classe de tipo em específico, similarmente em Haskell. Um método denominado *unbundle*.

Afim de mostrar que a permutação definida acima, implementa permutação mostramos que ela forma um grupo de permutação. Primeiro as classes de predicado e respectivas notações:

A definição **Group** pode ser lido da seguinte forma: o tipo **A** forma um grupo, se dado um elemento neutro, uma operação binária e uma operação unária inverso, ele satisfaz os axiomas padrão de grupo. Além disso queremos garantir que a relação seja uma equivalência, e que a operação binária e unária respeitem a equivalência (**grp_op_proper** e **grp_inv_proper**). Ao estabelecermos as propriedades **Proper**, indicamos ao Coq que é seguro fazer a reescrita no contexto das funções, desde que os termos reescritos sejam equivalentes. Dessa forma conseguimos demonstrar propriedades para as classes idênticas as provas tradicionais, simplificando a formalização: **TALVES SEJA MAIS INTERESSANTE MOSTRAR UM LEMMA DE AÇÃO DE GRUPO**

5 Perspectivas Futuras

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 na página 7.
- 2 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.
- 3 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.
- 4 COQUAND, T.; PAULIN, C. Inductively defined types. In: *COLOG-88*. [S.l.]: Springer Berlin Heidelberg, 1990. p. 50–66. Citado na página 11.
- 5 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.
- 6 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.
- 7 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 11.
- 8 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 13.
- 9 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 11.
- 10 BISHOP, E. *Foundations of Constructive Analysis*. [S.l.]: Ishi Press International, 2012. ISBN 4871877140. Citado na página 11.
- 11 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 12.
- 12 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 12.
- 13 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 12.

- 14 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 13.
- 15 Haskell Data.Monoid. <<https://web.archive.org/web/20210524082532/https://hackage.haskell.org/package/base-4.15.0.0/docs/Data-Monoid.html#g:3>>. Acessado: 28-05-2021. Citado na página 13.
- 16 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 13.
- 17 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 13.
- 18 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 13.
- 19 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 13.
- 20 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 15 e 16.
- 21 CHOUDHURY, P. *Constructive Representation of Nominal Sets in Agda*. Dissertação (Mestrado) — University of Cambridge, jun. 2015. Citado 2 vezes nas páginas 15 e 18.
- 22 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 15 e 16.
- 23 AMBAL, G.; LENGLET, S.; SCHMITT, A. $\text{HO}\pi$ in Coq. *Journal of Automated Reasoning*, Springer Science and Business Media LLC, v. 65, n. 1, p. 75–124, sep 2020. Citado na página 15.
- 24 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 15.
- 25 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 15.

- 26 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 15.
- 27 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 15.
- 28 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 18 e 19.
- 29 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 18.
- 30 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 18.
- 31 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 18.
- 32 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 18.
- 33 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.
- 34 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.
- 35 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.