Equipe abnTEX2

Modelo Canônico de Trabalho Acadêmico com abnTEX2

Brasil

2018, v<VERSION>

Equipe abnTEX2

Modelo Canônico de Trabalho Acadêmico com abnTEX2

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

Universidade do Brasil – UBr

Faculdade de Arquitetura da Informação

Programa de Pós-Graduação

Orientador: Lauro César Araujo

Coorientador: Equipe abnTEX2

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, antecedidas 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

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 -> Type), 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. São 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. Isso tudo torna vida do usuário um inferno. 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`.

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

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

3.3 Classes de tipos

Igualdade do coq e setoids

Classes de tipos

Teoria de tipos dependentes.

Termos convertíveis são iguais no Coq. Tipo indutivo eq. "=" é a igualdade padrão no Coq "≡" igualdade *setoid*. Pode-se substituir termos iguais em qualquer expressão pois eles são convertíveis (igualdade de Leibniz). Mas ao introduzir uma noção diferente de equivalência, aonde termos não convertíveis são equivalentes, limitamos a reescrita apenas somente a subtermos que são argumentos de funções que respeitam a equivalência, chamadas de funções próprias em relação a equivalência. A igualdade é "forte" demais, em alguns casos temos objetos que são conceitualmente iguais, mas sintaticamente diferentes. A igualdade no Coq é sintática. Coq não tem tipos quocientes (e não é uma boa ideia implementar PROCURAR REFERÊNCIA checagem de tipos indecidível), utiliza-se *setoids* (conjuntos de Bishop). A substituição de subtermos equivalentes é chamado de reescrita de setoids, e é necessário o gerenciamento correto da aplicação de lemas próprios de funções e equivalências. Sem uma infraestrutura adequada as provas começam a ficar incontroláveis e grandes, aonde surge o setoid hell. Coq possui um mecanismo que da suporte a reescrita setoid baseado em classes de tipos.

Falar sobre classes de tipos. É semelhante a implementação em Haskell (falar pq isso é legal tipos de classes), mas no Coq são cidadões de primeira classe graças a teoria de tipos dependentes. Classes são implementadas como records (registros) aliado a tipos implícitos e busca de provas. Registros em Coq são dependentes, ou seja, um membro do registro pode referenciar um membro anterior. Instâncias das classes são instâncias ordinárias dos registros, mas cada instância é registrada em um banco de dados de "sugestões" para a busca de provas (semelhante ao Haskell), entretanto Coq permite mais de uma instância de classe (contrário do Haskell), pois pode-se ter mais de uma instância de um registro.

DIFERENÇA ENTRE SET E PROP. FALAR REESCRITA GENERALIZADA E TIPOS DE CLASSES.

3.4 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.* (13), seção 4.2, e Choudhury (14), seção 4.3, ambos em Agda. A primeira formalização está relacionado a uma classe de implementações pragmáticas, também denominadas de "técnicas nominais" (13, 15, 16, 17, 18), enquanto a segunda, pertence ao grupo de formalizações de conjuntos nominais, ou sintaxe abstrata nominal (19, 14).

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 (20), 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 sentindo 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.* (13, 15) 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 (`αCompat`):

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

Para simplificação, omito 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 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 → 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 → 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.
Lemma LItStrongCompat {A} fvar fapp fabs L:
  αStrongCompat (@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} ∪ fv N ∪ fv M) in (* fv = support *)
```

```
  @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`, 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 Pritam (Quais titulos?)

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 algums 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 dividas 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:

```
Class Neutral A := neutral : A.        Notation ε := neutral.
Class Operator A := op: A → A → A.     Infix "+" := op.
Class Inverse A := inv : A → A.        Notation "- x" := (inv x).
```

E a definição de Grupo (omitindo parte da implementação):

```
Class Group (A : Type)
   `{Ntr: Neutral A, Opr: Operator A, Inv: Inverse A, Equiv A} : Prop := {
  grp_setoid :> Equivalence(≡@{A});
  grp_op_proper :> Proper ((≡@{A}) ⇒ (≡@{A}) ⇒ (≡@{A})) (+);
  grp_inv_proper :> Proper ((≡@{A}) ⇒ (≡@{A})) (-);

  grp_assoc : ∀ (x y z : A), x + (y + z) ≡@{A} (x + y) + z;
  (* ... *)
}.
Instance PermGrp: Group perm. Proof. (* ... *) Qed.
```

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 opera 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

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

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

```
Lemma grp_inv_involutive
  (x: G) `{Group G}: -(-x) ≡ 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.
```

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 na página 11.

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 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. Nenhuma citação no texto.

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 2 vezes nas páginas 15 e 16.

14 CHOUDHURY, P. *Constructive Representation of Nominal Sets in Agda*. Dissertação (Mestrado) — University of Cambridge, jun. 2015. Citado na página 15.

15 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.

16 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 na página 15.

17 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.

18 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.

19 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.

20 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.