UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPE RAMOS

PROVA DO ALGORITMO DE BRZOZOWSKI ASSISTIDA POR COMPUTADOR

JOINVILLE - SC

2021

FILIPE RAMOS

PROVA DO ALGORITMO DE BRZOZOWSKI ASSISTIDA POR COMPUTADOR

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Dra. Karina Girardi Roggia
Coorientador: Me. Rafael Castro Gonçalves Silva

JOINVILLE - SC

2021

RESUMO

RESUMO

Palavras-chaves: .

ABSTRACT

ABSTRACT

Keywords: .

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS E ABREVIATURAS

AFD Autômato finito determinístico

SED Sistema a eventos discretos

SUMÁRIO

1 INTRODUÇÃO

2 ASSISTENTE DE PROVAS

2.1 TEORIA DOS TIPOS INTUICIONISTA

2.1.1 Isomorfismo de Curry-Howard

Esta subseção visa mostrar como as provas escritas em lógica intuicionista correspondem a programas da linguagem funcional. Para tanto, utilizaremos a noção de tipos das linguagens de programação a fim de relacionar as proposições intuicionistas com estes. A relação que construiremos será a seguinte. Sejam P uma proposição qualquer e P' qualquer tipo correspondente. Deverá existir, então, uma prova para P se e somente se houver algum termo válido do tipo P'. Este termo será computado por um programa, razão pela qual designaremos como provas os programas cujo objetivo seja esta correspondência. A fins de simplificação, falaremos em existência de termos quando eles forem corretamente tipados; assim, poderemos poupar a palavra válido após qualquer frase de existência.

Inicialmente demonstremos que, para quaisquer proposições P e Q e respectivos tipos correspondentes P' e Q', existe uma prova para $P \star Q$ se e somente se há um termo válido do tipo $P' * Q'$. Aqui $\star$ é um conectivo lógico e $*$, o conjunto de todos os construtores para o tipo correspondente $P' * Q'$.

Supondo $\star = \vee$, podemos ter dois construtores para $* = \oplus$:

$$\oplus := \{\text{left}, \text{right}\}$$

em que left : $P' \to P' \oplus Q'$ é o construtor que recebe como entrada termo do tipo P' e right : $Q' \to P' \oplus Q'$ é o que recebe termo do tipo Q'. Esse isomorfismo é facilmente validado deste modo. ($\Rightarrow$) Por definição, $P \vee Q$ tem prova se (i) P ou (ii) Q têm prova. Assim, há (i) $p : P'$ ou (ii) $q : Q'$ e, portanto, (i) left(p) ou (ii) right(q). ($\Leftarrow$) Já se há $x : P' \oplus Q'$, então x é gerado por um dos construtores: (i) left(p) com algum $p : P'$ ou (ii) right(q) com algum $q : Q'$. Dessarte, há prova para (i) P ou (ii) Q, como queríamos demonstrar.

De maneira semelhante, para $\star = \wedge$ e $* = \otimes$:

$$\otimes := \{\text{and}\}$$

em que and : $P' \to Q' \to P' \otimes Q'$ é o construtor que recebe termos dos tipos P' e Q' respectivamente. ($\Rightarrow$) Se há prova para $P \wedge Q$, então também há para P e Q – existem, pois, $p : P'$ e $q : Q'$. Por conseguinte, and(p, q) é termo válido do tipo $P' \otimes Q'$. ($\Leftarrow$) Por

outro lado, quando há $x : P' \otimes Q'$, $x = \mathsf{and}(p, q)$ para algum $p : P'$ e algum $q : Q'$. Logo, deve existir prova para $P \wedge Q$.

No caso da implicação, o tipo correspondente é o funcional; ou seja, $P \Rightarrow Q$ é passível de prova se e somente se existe função do tipo $P' \to Q'$. Para entender essa correspondência, vale visualizar a estrutura de uma função $f : P' \to Q'$:

$$f(p : P') := \textbf{return } q : Q'$$

Toda função se constitui de um escopo, conjunto de todas as variáveis que são usadas ou não por ela para computar um resultado. Neste contexto, as variáveis[1] – que são termos corretamente tipados – correspondem a provas ou hipóteses de certas proposições. O escopo engloba as variáveis globais – que seriam os teoremas já provados e axiomas – e os argumentos. Assim, ao inserir-se um argumento, supõe-se um novo termo no escopo da função, o que equivale à noção de implicação ou suposição lógica.

Uma regra de inferência muito empregada na prova de teoremas é a eliminação da implicação ou *modus ponens*:

$$\frac{P \Rightarrow Q, P}{\therefore Q}$$

que é facilmente obtida na programação a partir da aplicação da função correspondente à prova de $P \Rightarrow Q$ sobre o termo de tipo correspondente de Q:

$$f : P' \to Q'$$
$$p : P'$$
$$f(p) : Q'$$

conforme havíamos definido.

A negação lógica $\neg P$ é definida a partir da implicação:

$$\neg P := P \Rightarrow \mathsf{F}$$

Outrossim, o tipo correspondente pode ser

$$\sim P' := P \to \mathsf{F}'$$

sendo F' um tipo vazio, sem construtores, o que significa

$$\nexists p : \mathsf{F}'$$

[1] Considerando apenas as variáveis atribuídas

Portanto, nunca haverá prova para a proposição correspondente, identicamente à proposição vazia.

Se quisermos adicionar, ainda, um tipo a corresponder com a proposição tautológica, bastará-nos definir um construtor sem argumentos; por exemplo, true : $\varnothing \to$ T′. Claramente, true$()$ serve de prova para a tautologia.

Utilizamos, muitas vezes, quantificadores lógicos em nossas proposições, como:

$$Q := \forall x : X, P(x)$$

em que P é uma proposição que depende do valor de x, sendo X um tipo qualquer. Seguindo nosso raciocínio, o correspondente de P deve ser um tipo dependente, da mesma maneira. Um termo de Q' seria da forma

$$f(x : X) := \textbf{return}\ p : (P'(x))$$

Sendo assim, f seria capaz de gerar um termo válido p do tipo que dependesse de qualquer argumento. Em outras palavras, f retornaria, para todo e qualquer $x : X$ de entrada, um termo do tipo dependente $P'(x)$ – o isomorfismo faz-se evidente. Aqui reside um possível questionamento: qual é o tipo de f? Explicitar isso visando a um entendimento fácil não é muito trivial, mas no CIC seria feito semelhantemente a

$$f : (\forall' x : X, P'(x))$$

estreitando a relação entre tal cálculo e a lógica intuicionista.

Já o quantificador existencial

$$Q := \exists x : X, P(x)$$

é demonstrado por meio de um $x : X$ qualquer e de uma prova de $P(x)$. Sejam $x_0 : X$ um termo qualquer, o construtor existencial

$$\text{exist} : (\forall' x : X, P'(x) \to \exists' x : X, P'(x)))$$

e $p : P'(x_0)$ o correspondente de prova para $P(x_0)$, o termo exist$(x_0)(p) : (\exists' x : X, P'(x))$ corresponde a uma prova de Q.

Uma relação importante e que estará presente no decorrer deste trabalho é a de igualdade. Possivelmente determinamos um construtor para o tipo correspondente dela destarte:

$$\text{refl}(x : X) : (x =' x)$$

novamente mediante um tipo dependente. Além disso, para demonstrar diversas igualdades e outras relações, precisamos da técnica de prova por indução, que, no presente isomorfismo, corresponde à recursão de funções. De modo a simplificar a elucidação, tenhamos de exemplo o tipo dos naturais ($\mathbb{N}$). Vamos representar esses números na forma unária, já que é fácil de compreender e implementar. Para tanto, usaremos dois construtores: um vazio e outro que recebe um natural (de maneira recursiva). Aquele representará o zero (0) e este ($S(n : \mathbb{N})$), o sucessor de n. Assim, na prática da nossa representação, teremos os termos da Tabela 1.

Representação unária	**Decimal**
0	0
$S(0)$	1
$S(S(0))$	2
$S(S(S(0)))$	3
$S(S(S(S(0))))$	4
$S(S(S(S(S(0)))))$	5
$\vdots$	$\vdots$

Tabela 1 – Naturais em representação unária e decimal

Agora, examinemos a função

$$\begin{aligned} f(p_0, p_n) :=\ & \mathsf{rec}(0) := p_0 \\ & \mathsf{rec}(S(n)) := p_n(n)(\mathsf{rec}(n)) \\ & \textbf{return } \mathrm{rec} \end{aligned}$$

em que $p_0 : P'(0)$ corresponde à hipótese de alguma proposição P sobre o zero e p_n é do tipo $\forall' n : \mathbb{N}, P'(n) \to P'(S(n))$. Dentro da função f, definimos outra função, rec, separando em casos distintos para cada construtor dos naturais. Observando os tipos acima, notamos que p_0 equivale à base da prova por indução e $p_n(n, h)$ para qualquer $n : \mathbb{N}$, ao passo indutivo, sendo h o equivalente à hipótese de indução.

Uma prova completa da correspondência de Curry-Howard vai além (SILVA, 2017). Não obstante, com o exposto, podemos captar as noções principais para o entendimento do funcionamento do assistente de provas Coq neste trabalho de conclusão de curso. Tal sistema computacional funciona como um verificador de provas ao passo que valida os programas que construímos (ou seja, provas).

2.1.2 Lei do terceiro excluído

2.2 O ASSISTENTE COQ

3 AUTÔMATOS FINITOS

Um autômato finito não-determinístico (AFND), ou simplesmente autômato finito (AF), é a abstração de uma máquina que decide uma linguagem regular. Para isso, ela conta com um conjunto de estados e regras de transição definida com base em um alfabeto, o conjunto de símbolos que são permitidos na entrada da máquina. Uma regra de transição indica os próximos estados da computação de uma palavra – sequência de símbolos – feita símbolo a símbol, da esquerda para a direita. O processo inicia-se nos estados iniciais e, quando termina em pelo menos um estado de aceitação, aceita a entrada. O nome não-determinístico serve para indicar a possibilidade de a computação ser não-determinística, no sentido de que pode ocorrer em mais de um estado simultaneamente. Por outro lado, um autômato finito determinístico (AFD) não compartilha dessa característica, havendo, portanto, apenas um estado inicial e regras de transição determinísticas: a partir de um estado e símbolo, só é possível transicionar para um único estado.

Veen, Rot e Geuvers (2007) definem um AF G como uma quíntupla

$$G = (\Sigma, S, I, \delta, F) \tag{3.1}$$

respectivamente o alfabeto, o conjunto de estados, o conjunto de estados inciais, o conjunto de regras de transição – ou função parcial de transição – e o conjunto de estados finais, ou de aceitação. Notadamente, essa definição vem com algumas restrições, conforme o Quadro 1. É interessante torná-las restrições intrínsecas ao tipo que definiremos em vez de utilizar condicionais, pois facilita as provas no Coq por alguns motivos: as restrições ficam implícitas e o compilador consegue verificá-las automaticamente.

Quadro 1 – Restrições da definição de um AFND G

	Componente	**Restrição**
1	S	Conjunto finito
2	Σ	
3	I	$I \subseteq S$
4	F	$F \subseteq S$
5	δ	$\delta : S \to \Sigma \to S$

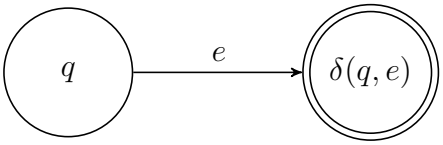
Fonte: Elaborado pelo autor, 2021.

Nas seções seguintes, discutimos a representação de AFs, outras definições relevantes a este contexto e algumas aplicações.

3.1 DIAGRAMA DE ESTADOS

Para auxiliar na visualização das transições entre os estados dos autômatos, os AFs são comumente representados por diagramas de estados. Nessa representação, os estados são nós de uma estrutura semelhante a de grafos, e as transições, arestas que interligam dois nós, conforme a Figura 1. Representam-se as transições cuja origem e destino são o mesmo estado por *loops*: arestas que partem de um nó e terminam no mesmo.

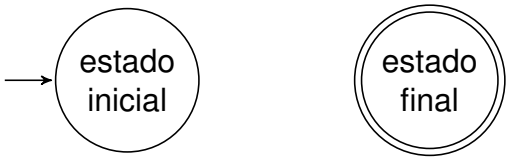
Figura 1 – Representação da transição de estados em um diagrama



Fonte: Elaborada pelo autor, 2021.

Nesta classe de diagramas, os estados inicial e final podem ser destacados de alguma forma. Para este trabalho, uma seta sem nó de origem aponta sempre para o nó do estado inicial, e uma circunferência dupla enfatiza o de um estado final, como demonstra a Figura 2.

Figura 2 – Representação de estados inicial (à esquerda) e final (à direita) em um diagrama



Fonte: Elaborada pelo autor, 2021.

Pode-se citar outros aspectos desta representação de autômatos: a possibilidade de adicionar rótulos aos nós, a opção de omitir os nomes dos estados nos nós quando não forem necessários e a aglutinação de transições que partem e terminam no mesmo estado em uma mesma aresta, com os símbolos separados por vírgula.

3.2 REPRESENTAÇÃO COMPUTACIONAL DE AUTÔMATOS FINITOS

Antes de iniciar qualquer prova sobre AFs no assistente de provas, devemos definir o que são esses autômatos na linguagem específica do sistema. Para tanto, precisamos utilizar estruturas de dados convencionais, pois estamos tratando de um assistente computacional. Novamente, facilita os trabalhos sequentes as restrições dos AFs representados pelas nossas estruturas de dados serem inerentes a elas.

Façamos uma singela modificação na definição da Equação 3.1. Fixemos

$$S = S' \cup I \cup F \cup \{s \mid \exists a s', (s, a, s') \in \delta \vee (s', a, s) \in \delta\}$$

sendo o nosso conjunto de estados definido agora pela união dos estados iniciais (I), finais (F), definidos pelas transições e outros possivelmente desconexos (S'). Além disso, passemos a definir o alfabeto como

$$\Sigma = \Sigma' \cup \{a \mid \exists s s', (s, a, s') \in \delta\}$$

de forma semelhante, a união dos símbolos contidos em transições e dos não utilizados (Σ'). Apesar de ser uma maneira aparentemente mais complexa para o nosso entendimento do que é um AF, essa nova forma substitui as restrições condicionais de forma conveniente. Desse modo, um AF G não será mais rotulado pela quíntupla da Equação 3.1, mas por cinco novos conjuntos de componentes:

- conjunto finito de estados (S');
- conjunto finito de estados iniciais (I);
- conjunto finito de estados finais (F);
- conjunto finito de símbolos (Σ');
- conjutno finito de transições (δ).

Essa nova representação define o alfabeto (Σ) e conjunto de estados (S) de forma implícita, nos restando apenas agora as restrições 1 e 2 do Quadro 1.

Agora nos resta representar os conjuntos finitos supracitados como estruturas de dados computacionais. Uma abordagem comum é o uso de *sets*, uma espécie de lista ordenada sem repetição (BLOT; DAGAND; LAWALL, 2016). A vantagem dessa estrutura é a extensionalidade: dois conjuntos são iguais se e somente se suas representações são iguais. Todavia, ela necessita de comparadores, e os algoritmos de ordenamento interferem no desempenho da computação, entre outros. No presente estudo, a ausência da propriedade extensional não obsta as provas sobre AFs uma vez que podemos substituir a relação de igualdade de conjuntos por uma nova relação de equivalência. Assim, se representarmos os conjuntos por listas simples, duas listas $L1$ e $L2$ serão ditas equivalentes se

$$\forall x, x \in L1 \leftrightarrow x \in L2$$

permitindo inclusive elementos fora da ordem ou repetidos nas estruturas.

Figura 3 – Estrutura de dados para AFs

Fonte: Elaborada pelo autor, 2021.

```
Inductive nfa_comp {A B} :=
| state (q:A)
| symbol (a:B)
| start (q:A)
| accept (q:A)
| trans (q:A) (a:B) (q':A).

Definition nfa_comp_list A B := list (@nfa_comp A B).
```

Baseado nisso, a estrutura usada por este trabalho, já na linguagem do assistente de provas Coq, é a que segue na Figura 3. É importante notarmos uma nova restrição decorrente da teoria dos tipos – implementada pelo Coq. Antes falávamos em elementos quaisquer, diferenciando-os apenas pelos conjuntos aos quais pertenciam. Ao utilizarmos tipos, no entanto, condicionamos os elementos – agora termos – a eles. Os termos podem ser, por exemplo, números naturais, strings ou de qualquer tipo que definirmos. Nem todos os tipos, entretanto, nos garantem uma propriedade essencial para a computação dos algoritmos sobre os AFs representados: a decidibilidade da igualdade. Um tipo A tem igualdade decidível se existe algum decisor f tal que

$$\forall xy, f(x, y) \leftrightarrow x = y$$

Dessarte, junto aos componentes, nossa representação abarcará a restrição de existência desses decisores para os tipos `A` e `B` da Figura 3.

3.3 FUNÇÃO DE TRANSIÇÃO

Uma das funções mais importantes relativas aos AFs é a de transição. A função de transição δ de um autômato G é tal que

$$\forall sas', s' \in \delta(s, a) \leftrightarrow \texttt{trans}\ s\ a\ s' \in G$$

Ela pode ser estendida para que funcione computando uma dada sequência de símbolos (lista) w a partir de um dado conjunto de estados Q desta maneira

$$\delta(Q, \epsilon) = Q$$

$$\delta(Q, a) = \{s \mid \exists s' a, \texttt{trans}\ s'\ a\ s \in G \wedge s' \in Q\}$$

$$\delta(Q, aw) = \delta(\delta(Q, a), w)$$

em que ϵ é a palavra vazia e $a \in \Sigma$ é um símbolo do alfabeto do autômato. Desenvolvemos tal função facilmente no Coq, de forma segmentada. Definimos, por exemplo, uma função que recebe um estado e símbolo e, por meio de sucessivas iterações na lista que o representa, constrói uma lista de estados que são transicionados a partir daqueles; depois escrevemos uma função estendida que se utiliza dessa última para generalizar as entradas para listas de estados e símbolos. Essas funções também recebem os decisores de igualdade do AF, que suprimiremos no decorrer do presente trabalho.

A função de transição é responsável pela computação da entrada que o autômato recebe. Temos a possibilidade de formalizar a linguagem do AF G como o conjunto

$$L(G) = \{w \mid \exists s, s \in F \land s \in \delta(I, w)\}$$

Logo, para verificar se uma palavra é aceita pelo AF, basta-nos rodar a função de transição sobre os estados iniciais e essa dada palavra e verificar se um dos termos resultantes é estado pertencente ao conjunto de estados finais.

Outra definição importante relacionada à de transições tange aos caminhos. Essa é uma definição indutiva de dois construtores:

- para todo possível estado[1] s, existe um caminho de s a s por ϵ;
- para todos os estados s_1, s_2, símbolo a e palavra w, se existe uma transição $\texttt{trans}\ s_2\ a\ s_3 \in G$ e um caminho de s_1 a s_2 por w, então existe um caminho de s_1 a s_3 por wa.

A noção de caminho é relevante para tratar de buscas[2].

3.4 ALCANÇABILIDADE DOS ESTADOS

3.5 REVERSÃO

3.6 AUTÔMATOS FINITOS DETERMINÍSTICOS

Os autômatos finitos determinísticos (AFDs) carregam duas restrições a mais: (1) possuem apenas um estado inicial e (2) as transições são determinísticas, ou seja, não existe uma transição partindo de um mesmo estado e mediante um mesmo símbolo para um estado destino diferente. Consideremos permitida a existência de AFDs

[1] Termo do tipo dos estados do autômato
[2] Como a busca em profundidade

sem estado inicial a fim de simplificar o presente estudo. Isso não acarretará inconsistência, uma vez que a representação de um AFD sem estado inicial é equivalente à de um AFD com um estado inicial sem transições a partir dele – a linguagem e as propriedades de interesse do trabalho se mantêm isomórficas. Para representar os AFDs no Coq, utilizemos a mesma estrutura de dados e um tipo proposição indutivamente definido. Basicamente, esse tipo receberá construtores para cada construtor de AF, com diferenças para os construtores de estado inicial e transição. Nesses casos, adicionamos hipóteses para garantir as restrições.

Um modo de garantir a restrição (1) é criando dois construtores. O primeiro receberá um AF G_1 qualquer, uma prova de que ele é determinístico, uma prova de que ele não possui estados inicias e um estado inicial q_0. Dessa forma podemos afirmar que o o autômato $\texttt{start}\ q0 :: G_1$ é determinístico. O outro construtor receberá um AF G_2 qualquer, uma prova de que ele é determinístico, um estado inicial q_1 e uma prova de que já existe um estado inicial igual a q_1. Outrossim, o autômato $\texttt{start}\ q_1 :: G_2$ também é determinístico.

Semelhantemente, em relação à restrição (2), podemos utilizar a mesma tática. Primeiro garantimos que não existe uma transição $\texttt{trans}\ q\ a\ s$ para todo e qualquer s no AFD G_1 e obtemos um AFD $\texttt{trans}\ q\ a\ q' :: G_1$. Depois definimos que existe $\texttt{trans}\ r\ b\ r'$ no AFD G_2 para termos um AFD $\texttt{trans}\ r\ b\ r' :: G_2$.

3.7 DETERMINIZAÇÃO

3.8 APLICAÇÕES

4 MINIMIZAÇÃO DE AUTÔMATOS FINITOS DETERMINÍSTICOS

4.1 ALGORITMO DE BRZOZOWSKI

5 PROVA DO ALGORITMO DE BRZOZOWSKI

5.1 PROVAS SOBRE REVERSÃO

5.2 PROVA DO ALGORITMO DE DETERMINIZAÇÃO

6 CONSIDERAÇÕES FINAIS

REFERÊNCIAS

BLOT, A.; DAGAND, P.-É.; LAWALL, J. From sets to bits in coq. In: SPRINGER. **International Symposium on Functional and Logic Programming**. [S.l.], 2016. p. 12–28. Acesso em: 9 jul. 2021.

SILVA, R. C. G. **Visão Categórica do Sistema de Tipos de Haskell**. Monografia (Trabalho de Conclusão de Curso) — Bacharelado em Ciência da Computação, Universidade do Estado de Santa Catarina, Joinville, 2017. Disponível em: <http://sistemabu.udesc.br/pergamumweb/vinculos/000043/000043b8.pdf>. Acesso em: 12 mar. 2021.

VEEN, E. van der; ROT, J.; GEUVERS, J. The practical performance of automata minimization algorithms. **Comparative Study of Performance of Tabulation and Partition**, v. 27, 2007. Disponível em: <http://www.cs.ru.nl/bachelors-theses/2017/Erin_van_der_Veen___4431200___The_Practical_Performance_of_Automata_Minimization_Algorithms.pdf>. Acesso em: 8 jul. 2021.