UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC

CENTRO DE CIÊNCIAS TECNOLÓGICAS - CCT

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPE RAMOS

PROVA DA MINIMIZAÇÃO DE AUTÔMATOS FINITOS DETERMINÍSTICOS PELO ALGORITMO DE BRZOZOWSKI ASSISTIDA POR COMPUTADOR

JOINVILLE - SC

2021

FILIPE RAMOS

PROVA DA MINIMIZAÇÃO DE AUTÔMATOS FINITOS DETERMINÍSTICOS PELO 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

Filipe Ramos

Prova da minimização de autômatos finitos determinísticos pelo 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.

Banca examinadora:

Orientadora: _____

Dra. Karina Girardi Roggia
Udesc

Membros: _____

Dr. Cristiano Damiani Vasconcellos
Udesc

_____

Dr. Roberto Silvio Ubertino Rosso Junior
Udesc

_____

Dr. Yuri Kaszubowski Lopes
Udesc

Joinville, 27 de agosto de 2021

RESUMO

Os autômatos finitos, reconhecedores de linguagens regulares, são muito importantes para a ciência da computação e o estudo de sistemas orientados a eventos discretos e assíncronos na automação industrial. A classe de autômatos finitos determinísticos tem destaque na computação prática ao passo que está mais próxima dos cálculos determinísticos que um computador executa. Nesse contexto, podemos obter otimização na representação e operação dessas máquinas por meio da minimização dos autômatos, achatando as estruturas de dados com a diminuição do número de estados. Um computador também pode auxiliar no processo de demonstração de propriedades sobre algoritmos, apresentando artifícios facilitadores e mecanismos de validação, razão por que é relevante a sua aplicação na prova do corretismo da minimização de Brzozowski. O presente trabalho apresenta demonstrações assistidas por computador para mostrar que tal algoritmo funciona.

Palavras-chaves: algoritmo de Brzozowski, autômatos finitos, minimização de autômatos finitos determinísticos, assistente de provas, Coq.

ABSTRACT

Finite automata–regular languages recognizers–are very important for computer science and the study of discrete and asynchronous event-oriented systems in industrial automation. The class of deterministic finite automata stands out in practical computing as it is connected to the deterministic calculations that a computer performs. In this context, we can optimize the representation and execution of these machines by minimizing the automata, shortening the data structures through a reduction in the number of states. Moreover, a computer can help in the process of demonstrating properties of algorithms by providing some tools and validation mechanisms–the reason why its application is relevant in the correctness proof of Brzozowski's minimization. This work presents computer-assisted demonstrations showing that this algorithm is correct.

Keywords: Brzozowski algorithm, finite automata, deterministic finite automata minimization, proof assistant, Coq.

LISTA DE ILUSTRAÇÕES

Figura 1 – Representação da transição de estados em um diagrama 22
Figura 2 – Representação de estados inicial e final em um diagrama 22
Figura 3 – Estrutura de dados para AFNs 24
Figura 4 – Diagrama de um AFN qualquer 26
Figura 5 – AFN com estados inalcançáveis 28
Figura 6 – Protocolo para um canal de redes de computadores 35
Figura 7 – Aglutinação de caminhos 37
Figura 8 – Observação básica sobre o algoritmo de Brzozowski 46
Figura 9 – Lema do bombeamento 58
Figura 10 – Princípio da casa dos pombos 59

LISTA DE SIGLAS E ABREVIATURAS

AFD Autômato finito determinístico

AFN Autômato finito não determinístico

SED Sistema a eventos discretos

SUMÁRIO

1 INTRODUÇÃO 13

2 ASSISTENTE DE PROVAS 17
2.1 TEORIA DOS TIPOS INTUICIONISTA 18
2.2 O ASSISTENTE COQ 19

3 AUTÔMATOS FINITOS 21
3.1 DIAGRAMA DE ESTADOS 21
3.2 REPRESENTAÇÃO COMPUTACIONAL 22
3.3 FUNÇÃO DE TRANSIÇÃO 26
3.4 ALCANÇABILIDADE DOS ESTADOS 28
3.5 AUTÔMATOS FINITOS DETERMINÍSTICOS 29
3.6 EQUIVALÊNCIA DE ESTADOS 30
3.6.1 Equivalência de autômatos finitos determinísticos 31
3.7 REVERSÃO 32
3.8 DETERMINIZAÇÃO 33
3.9 APLICAÇÕES 34

4 MINIMIZAÇÃO DE AUTÔMATOS FINITOS DETERMINÍSTICOS . . 37
4.1 MINIMIZAÇÃO PELA UNIÃO DOS ESTADOS EQUIVALENTES . . . 37
4.2 ALGORITMO DE BRZOZOWSKI 38

5 TRABALHOS RELACIONADOS 41

6 ALGORITMO DE BRZOZOWSKI É CORRETO 43
6.1 REVERSÃO DA QUALIDADE DOS ESTADOS 48
6.2 LINGUAGEM REVERTIDA É REVERSA 50
6.3 ALGORITMO DE DETERMINIZAÇÃO 51
6.3.1 Normalização 53
6.3.2 Estados finais 54

6.3.3 Alcançabilidade dos estados 54
6.3.4 Determinismo 56
6.3.5 Relação entre caminhos 56
6.4 CONSIDERAÇÕES DO CAPÍTULO 59

7 CONSIDERAÇÕES FINAIS 61

REFERÊNCIAS 63

1 INTRODUÇÃO

Os autômatos são modelos de máquinas abstratas muito importantes para os estudos da computação teórica. Em particular, os autômatos finitos (AFNs) reconhecem as linguagens regulares, aplicadas na ciência da computação e tecnologia da informação (HOPCROFT; MOTWANI; ULLMAN, 2006). Sob outra perspectiva, essa classe de autômatos consegue descrever sistemas do mundo contemporâneo na automação de indústrias e tecnologias, já que o advento dos sistemas a eventos discretos (SEDs) impôs um novo paradigma, contrário aos sistemas orientados pelo tempo, que são, em geral, muito bem descritos pelos modelos físicos clássicos e modernos (CASSANDRAS; LAFORTUNE, 2008).

Os autômatos finitos determinísticos (AFDs) têm algumas restrições associadas ao determinismo; sabemos, a partir da entrada, das regras de transição e do estado inicial, determinar o último estado da computação da máquina. A determinização[1], por vezes espontânea, corrobora a importância dos AFDs e da otimização de suas estruturas de dados e demais representações. Para tanto, precisamos minimizar o AFD, o que implica em construir uma nova máquina capaz de reconhecer a mesma linguagem e dotada de menos estados, o que reduz a complexidade de espaço e ajuda a tornar o autômato mais inteligível. O algoritmo de Brzozowski é uma técnica que costuma ser eficiente para esse intuito, apesar de ter complexidade exponencial no pior caso (BONCHI et al., 2012). A minimização que Brzozowski e Tamm (2014) propõem envolve a reversão das transições, a transformação de estados inciais em finais e viceversa e posterior determinização, que pode ser complexa. Uma das abordagens para essa operação ocorre por meio de busca similar à de grafos, que otimiza a execução geral do algoritmo de Brzozowski. Os trabalhos relacionados efetuam a determinização pela construção do conjunto das partes dos estados do AFD, o que resulta em um número exponencial de estados, com a necessidade de analisar a existência de estados inalcançáveis para então removê-los. Por isso, o presente estudo é relevante ao utilizar a abordagem de busca para demonstrar o funcionamento do algoritmo de minimização sem análise de estados inalcançáveis. O emprego da assistência computacional eleva o valor do trabalho ao passo que tem validadas suas proposições.

[1] Neologismo para a obtenção de um AFD que executa as mesmas tarefas do AFN original

Além disso, pela similaridade dos algoritmos citados com outros métodos da teoria da computação, este estudo serve de inspiração para trabalhos futuros.

A constatação de propriedades sobre AFDs específicos pode não ser uma tarefa trivial, e sua demonstração costuma ser ainda mais difícil, pois a intuição, muitas vezes, é o único meio utilizado para a garantia dessas propriedades. Outra técnica utilizada com esse fim é o *model checking*, no qual os instrumentos de verificação – *model checkers* – operam enumerando exaustivamente o espaço dos estados de um modelo. O método é limitante, porque resulta em uma explosão combinatorial, tornando impraticável a verificação de modelos de sistemas reais desse modo (ATHALYE, 2017). Por isso, o emprego de assistentes interativos de provas matemáticas é útil, eles possibilitam a formulação e demonstração das propriedades. Tais assistentes reúnem meios formais e computacionais que propiciam a verificação dos passos da demonstração, auferindo rigor. Entre os assistentes de provas, o Coq é destaque por ter comunidade extensa, uma linguagem funcional de ordem superior e permitir a construção de tipos dependentes. Outras qualidades desse assistente são a possibilidade de programar táticas de provas, desenvolver e carregar módulos separadamente sem a necessidade de verificá-los novamente e definir notações para melhorar a visualização das definições e provas (BARRAS et al., 1999). O Coq tem se mostrado uma ferramenta robusta e bastante aceita pela comunidade acadêmica. A exemplo disso, Gonthier et al. (2008) desenvolveu, nessa plataforma, uma prova do teorema das quatro cores, que requer análise de um número de casos impraticável pelo ser humano por si só. O resultado do trabalho foi uma prova mais rigorosa em relação às demais demonstrações para o mesmo teorema, pois, embora elas também tenham sido assistidas por computador, esta tem o diferencial de dispensar a necessidade de confiar em programas verificadores de casos particulares e ser expressa na linguagem formal do Coq. Essas e outras características motivaram a escolha do Coq para a assistência das provas que este trabalho externa.

A partir do exposto, o objetivo geral deste trabalho de conclusão de curso consistiu em provar, mediante a assistência do sistema de provas Coq, o corretismo do algoritmo de Brzozowski. Os objetivos específicos foram:

1. implementar estruturas de dados e definições para representar AFNs na linguagem do assistente;

2. demonstrar que o algoritmo de Brzozowski retorna um AFD;
3. provar a característica reversa da linguagem do autômato resultante;
4. demonstrar que todos os estados desse AFD são alcançáveis;
5. provar a inexistência de estados indistinguíveis.

Com a finalidade de atingir os objetivos supracitados, o presente trabalho é estruturados conforme segue. Primeiramente, o Capítulo 2 introduz o assistente de provas Coq. No Capítulo 3, tratamos das definições de AFNs e sua representação no Coq. Depois, reduzimos nosso estudo sobre AFNs ao escopo da minimização, no Capítulo 4. Já no Capítulo 5, apresentamos os trabalhos relacionados a este estudo. O capítulo seguinte, 6, explana o desenvolvimento das provas que o presente trabalho realizou. Por fim, expressamos as considerações finais e, após, estão as referências bibliográficas.

Os artefatos que este trabalho produziu foram os seguintes arquivos de código do Coq:

- `utils.v`: estruturas de dados para listas e conjuntos, bem como lemas sobre as definições;
- `nfa.v`: definições e lemas gerais sobre AFNs;
- `dfa.v`: definições e lemas sobre AFDs;
- `reversing.v`: algoritmo de reversão de AFN e provas acerca da linguagem revertida;
- `normalizing.v`: algoritmo de normalização de estados conjuntos, assim como provas sobre diversas propriedades dos AFNs normalizados;
- `determinizing.v`: funções de determinização e lemas de seu corretismo;
- `brzozowski.v`: o algoritmo de Brzozowski e a prova de sua minimização.

Esses arquivos estão disponíveis em:

<https://github.com/fil1pe/brzozowski-algorithm>

2 ASSISTENTE DE PROVAS

Um assistente de provas interativo é um artifício de software que auxilia no processo de prova matemática. Diferentemente dos provadores automáticos de teoremas, os assistentes não realizam provas automaticamente com um simples comando do usuário, mas participa com ele no processo oferecendo uma variedade de procedimentos formalizadores. Justificamos seu uso também na necessidade de validar demonstrações, haja vista que detalhes podem passar despercebidos pelo olhar analítico do ser humano. Os assistentes mais robustos implementam a teoria dos tipos, na qual os elementos pertencem a tipos definidos recursivamente, e não a conjuntos.

Há pelo menos 13 provadores de teoremas disponíveis, entre os quais estão os provadores semiautomáticos, ou assistentes de provas. Os assistentes mais utilizados são: Agda, Coq, HOL, IMPS, Isabelle, Lego, Lean, Metamath, Mizar, NuPRL, Omega-MEGA, Phox e PVS. Informações acerca dessas ferramentas são consultáveis em Barendregt e Geuvers (2001), Moura et al. (2015). Em relação aos sistemas de tipos, o Metamath é o único da lista que não é tipado e utiliza lógica quântica, sem ligação com o presente trabalho. Em geral, os demais sistemas de provas apresentam características que nos permitem formular demonstrações sobre autômatos finitos. O Coq, enfoque do presente trabalho, é vantajoso por utilizar lógica de ordem superior, implementar a teoria dos tipos intuicionista (vide Seção 2.1) e permitir tipos dependentes, além de possuir um sistema de táticas que facilita o entendimento do código e o desenvolvimento. A teoria dos tipos fornece o encontro da ciência da computação com a matemática e a lógica. Ela é simultaneamente um sistema formal e linguagem de programação funcional que permite raciocinar enquanto se programa (LUO, 1994). Sob outra perspectiva, a escolha do provador tem base na subjetividade e no relativo conforto do usuário; afinal, o assistente de provas, como o nome sugere, tem o papel de auxiliar o usuário no processo de demonstrar teoremas. Os materiais didáticos acerca deste assistente também são um diferencial para esta opção (PIERCE et al., 2010).

2.1 TEORIA DOS TIPOS INTUICIONISTA

A teoria dos tipos intuicionista emprega artifícios lógicos inerentes não à lógica clássica, mas à construtivista. Imprescindível na programação, os tipos descrevem características das variáveis e constantes e o comportamento desses valores nas operações e comparações da linguagem utilizada. Devido a isso, é natural que usemos a teoria dos tipos para programar demonstrações em computador. Na prática, assistentes de provas simplesmente validam programas desenvolvidos em suas linguagens, conforme as técnicas de compiladores. Por isso, utilizamos a correspondência de Curry-Howard a fim de estabelecermos um isomorfismo entre provas e programas. Uma prova para uma proposição `P` é um termo do tipo `P` (SILVA, 2017; PIERCE et al., 2010).

Peça importante desse isomorfismo, a noção de construtores existe também no paradigma orientado a objeto (STROUSTRUP, 1988), e eles servem para construir dados (termos) e tipos. Podemos definir uma disjunção `P \/ Q` como um tipo de dois construtores: um recebe uma prova de `P` e o outro, de `Q`. A conjunção `P /\ Q` demonstramos por meio do seu único construtor aplicado a ambas as provas de `P` e `Q`. No caso da implicação, o tipo correspondente é o funcional; ou seja, `P -> Q` é o tipo de uma função que recebe uma prova `p:P` e retorna outra prova para `Q`. Quando aplicamos uma função sobre um outro termo, estamos fazendo uma inferência por *modus ponens*. Além disso, a recursão corresponde à prova por indução. Já os quantificadores descrevem funções de tipos dependentes, já que `f:∀ x:X, P x` é uma função que recebe um termo `x` do tipo `X` e retorna um termo cujo tipo depende de `x`.

Neste contexto, uma linguagem ou problema ser decidível significa existir algum algoritmo que resolve o problema para toda e qualquer entrada. Assim, é necessário que esse algoritmo seja uma função total, e não parcial, o que possibilitaria respostas indefinidas. O assistente de provas Coq impõe que todas as funções definidas em sua linguagem sejam totais; ou seja, toda função deve ter algum retorno, sem que exista alguma entrada que leve a uma computação de duração infinita dessa função. Isso é uma condição restritiva, mas indispensável para a prova de decidibilidade. Por exemplo, muito importante para o presente trabalho, a igualdade é um desses problemas que requerem, diversas vezes, a decidibilidade. Se uma função total responde

corretamente a “ `x1 = x2` ?”, dadas quaisquer entradas `x1` e `x2` de um tipo `A` , dizemos que ela decide a igualdade do tipo `A` . A teoria dos tipos intuicionista não possui a lei do terceiro excluído (RICHMAN, 1996). Dessa forma, a decidibilidade é importante, além do mais, para verificar se a regra do terceiro excluído

```
∀ x:A, (P x \/ ~ P x)
```

vale para o tipo `A` e a função `P` que determinam a linguagem a ser decidida. Isso é imprescindível para as provas por contradição, que partem do princípio de que uma proposição é falsa ou verdadeira e julgam que, se a proposição não pode ser falsa, ela deve ser verdadeira (ARISTÓTELES, 1985; MENEZES, 2000).

2.2 O ASSISTENTE COQ

O presente trabalho desenvolveu as demonstrações no assistente de provas Coq, motivo pelo qual a teoria dos tipos intuicionista, com suas particularidades, baseia as definições dos capítulos que seguem. A justificativa para esta escolha reside na ampla comunidade de desenvolvedores estabelecida, o que também confere muitas informações acessíveis em fóruns e meios científicos. Além disso, as demais facilidades de software – como a interface de desenvolvimento do sistema – são vantagens que fizeram efeito no resultado obtido pelo trabalho.

As especificidades da teoria dos tipos supracitadas são importantes para o entendimento de como ocorreu a evolução dos artefatos do presente trabalho. Além disso, a base e as motivações expostas neste capítulo suscitam os próximos capítulos.

3 AUTÔMATOS FINITOS

Um autômato finito não-determinístico (AFN), ou simplesmente autômato finito, é a abstração de uma máquina que decide uma linguagem regular. Para isso, ela conta com um conjunto de estados e regras de transição definidas 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 símbolo, da esquerda para a direita. O processo inicia 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 alguma transição do autômato pode levar a múltiplos distintos estados. “Um autômato finito não determinístico tem o poder de estar em vários estados ao mesmo tempo”, segundo Hopcroft, Motwani e Ullman (2006, p. 55, tradução do autor). 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.

É comum descrevermos matematicamente um AFN como uma quíntupla de conjuntos. Veen, Rot e Geuvers (2007) escrevem um AFN G desta forma:

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

respectivamente o alfabeto, o conjunto de estados, o conjunto de estados iniciais, o conjunto de regras de transição – ou função parcial de transição – e o conjunto de estados finais, ou de aceitação.

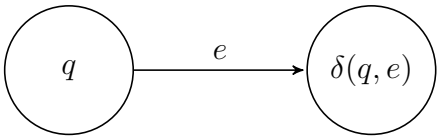
Nas seções seguintes, discutimos a representação de AFNs, outras definições relevantes a este contexto e algumas aplicações.

3.1 DIAGRAMA DE ESTADOS

Para auxiliar na visualização das transições entre os estados dos autômatos, os AFNs 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. Representamos as transições

cuja origem e destino são o mesmo estado por *loops*: arestas que partem de um nó e terminam no mesmo.

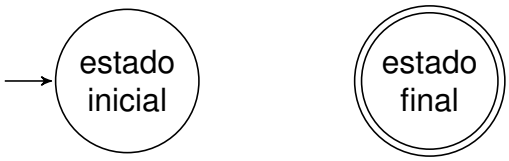
Figura 1 – Representação da transição de estados em um diagrama



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 e final em um diagrama



Fonte: Elaborada pelo autor, 2021.

Podemos 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

Antes de iniciar qualquer prova sobre AFNs no assistente de provas, devemos definir o que são os AFNs na linguagem específica do sistema. Para tanto, precisamos utilizar estruturas de dados convencionais, pois estamos tratando de um assistente computacional. Notadamente, a definição da Equação 3.1 vem com algumas restrições:

1. Σ e S são conjuntos finitos;
2. $I, F \subseteq S$;

3. δ é uma função do tipo $S \times \Sigma \to S$.

É interessante tornar essas 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. Façamos uma singela modificação na definição da Equação 3.1. Fixemos

$$S = Q \cup I \cup F \cup T \tag{3.2}$$

sendo o nosso conjunto de estados definido, implicitamente, agora, pela união dos estados iniciais (I), finais (F), dos que são definidos pelas transições:

$$T = \{s \mid \exists a, (\delta(s, a) \neq \varnothing \vee (\exists s', s \in \delta(s', a)))\} \tag{3.3}$$

e outros quaisquer (Q), possivelmente desconexos – isto é, sem transições partindo ou chegando a eles. Além disso, passemos a definir o alfabeto como

$$\Sigma = \Gamma \cup V \tag{3.4}$$

de forma semelhante, a união dos símbolos rotulados pelas transições:

$$V = \{a \mid \exists s, \delta(s, a) \neq \varnothing\} \tag{3.5}$$

e de outros quaisquer, como os não utilizados em nenhuma transição (Γ). Apesar de ser uma maneira aparentemente mais complexa para o nosso entendimento do que é um AFN, essa nova forma substitui as restrições condicionais de forma conveniente. Desse modo, um AFN G não será mais rotulado pela quíntupla da Equação 3.1, mas por cinco novos conjuntos de componentes:

- conjunto finito de estados qualquer (Q);
- conjunto finito de estados iniciais (I);
- conjunto finito de estados finais (F);
- conjunto finito de símbolos qualquer (Γ);
- conjunto finito de transições (δ).

Essa nova representação define o alfabeto (Σ) e conjunto dos estados (S) de forma que as regras que usamos na lógica clássica de Hopcroft, Motwani e Ullman (2006), Cassandras e Lafortune (2008) e Veen, Rot e Geuvers (2007) ficam implícitas na definição, restando-nos apenas a restrição 1 listada acima. A noção clássica de AFN deixa livre para o usuário da teoria a designação direta dos tais conjuntos, mas ele precisa respeitar todas as restrições impostas. Nosso objetivo é impedir o assistente de provas de considerar formulações que não sejam AFNs, permitindo a construção indireta desses conjuntos, que vai seguir o padrão que desejamos.

Agora nos resta representar os conjuntos finitos supracitados como estruturas de dados computacionais. Uma abordagem comum para a representação de conjuntos faz 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, os algoritmos de ordenamento interferem no desempenho da computação, entre outros. Outra dificuldade inerente aos *sets* reside na tentativa de representar conjuntos de conjuntos; ou seja, *sets* dentro de *sets*, uma necessidade do presente trabalho. Sob outra perspectiva, para nossos objetivos, a ausência da propriedade extensional não obsta as provas sobre AFNs, 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

```
∀ x, (x ∈ L1 <-> x ∈ L2)
```

permitindo inclusive elementos fora da ordem ou repetidos nas estruturas.

Figura 3 – Estrutura de dados para AFNs

```
Inductive nfa_comp {A B} :=
| state (q:A)
| symbol (a:B)
| start (q:A)
| accept (q:A)
| trans (q1:A) (a:B) (q2:A).

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

Fonte: Elaborada pelo autor, 2021.

Baseado nisso, a estrutura usada por este trabalho, já na linguagem do assistente de provas Coq, é o tipo `nfa` que consta 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 AFNs representados: a decidibilidade da igualdade. Um tipo `A` tem igualdade decidível se existe algum decisor `f` para a igualdade dos termos de `A`. Em outras palavras, precisa existir uma função `f:A->A->bool` que retorna `true` quando as entradas são iguais e `false` senão. Dessarte, junto aos componentes, nossa representação abarcará a restrição de existência desses decisores para os tipos `A` e `B` da Figura 3. Nos teoremas apresentados por este trabalho, tais decisores e as condicionais de seu corretismo nós omitiremos para fins de simplificação.

A fim de compreender as estruturas apresentadas, analisemos os seguintes exemplos:

1. `[start 0; trans 0 1 1; trans 0 1 0; trans 1 0 2] ;`
2. `[start 0; trans 0 1 1; trans 0 1 0; trans 1 0 2; trans 0 1 0; trans 1 0 2] ;`
3. `[start 0; trans 0 1 1; trans 1 0 2; trans 0 1 0] ;`
4. `[start 0; trans 0 1 1; trans 0 1 0; trans 1 0 2; state 0; symbol 2] .`

Todos os AFNs têm a mesma linguagem, embora a ordem do autômato 3 esteja trocada, o 2 tenha repetições e o 4 tenha um símbolo a mais que não é utilizado pela função de transição. O componente `state 0` do AFN (4) é inútil, pois o estado `0` já está definido pelas regras de transição `trans 0 1 1` e `trans 0 1 0`. Entretanto, é o `symbol 2` que difere este dos demais, já que o faz possuir um alfabeto diferente. De forma clássica, os três primeiros são representações diferentes para o mesmo AFN

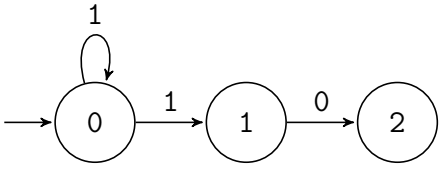
$$(\{0,1\},\{0,1,2\},\{0\},\{(0,1,1),(0,1,0),(1,0,2)\},\varnothing) \tag{3.6}$$

e o quarto descreve o AFN

$$(\{\,0\,,\,1\,,\,2\,\},\{\,0\,,\,1\,,\,2\,\},\{\,0\,\},\{(\,0\,,\,1\,,\,1\,),(\,0\,,\,1\,,\,0\,),(\,1\,,\,0\,,\,2\,)\},\varnothing) \tag{3.7}$$

O diagrama de ambos é o da Figura 4.

Figura 4 – Diagrama de um AFN qualquer



Fonte: Elaborada pelo autor, 2021.

Se, em vez de `state 0`, tivéssemos posto `state 3` no AFN (4), teríamos a representação para um autômato com um estado desconexo (`3`), já que não há um caminho do estado inicial `0` até ele. É sobre caminhos e transições que versa a Seção 3.3.

3.3 FUNÇÃO DE TRANSIÇÃO

Uma das funções mais importantes relativas aos AFNs é a de transição. Ela dita a computação das entradas do autômato, definindo as regras de transição de estados. No presente trabalho, entretanto, o contrário ocorre: as regras de transição ditam o funcionamento da função de transição. Nesse sentido, uma construção da função de transição δ deve ser tal que

```
∀ a q q' G, (q' ∈ (δ G q a) <-> trans q a q' ∈ G)
```

Ou seja, aplicada a função δ sobre dois termos `q` e `a`, o resultado terá um estado `q'` se houver uma regra de transição do estado `q` para o `q'` pelo símbolo `a`. Pensemos na computação dessa função como sucessivas iterações sobre os componentes do autômato de modo que seja feita uma busca por alguma regra de transição partindo de `q` por `a`.

A função de transição pode ser estendida para que funcione para uma dada sequência de possíveis símbolos[1] (lista) `w` a partir de um dado conjunto de estados `Q` desta maneira:

[1] Termos que não necessariamente fazem parte do alfabeto

```
Fixpoint δ̂ G Q w :=
  match w with
  | nil => Q
  | a::w' => δ̂ G (g G Q a) w
  end
```

em que `Fixpoint` indica a definição de função recursiva, `nil` , ou `[]` , é o construtor de lista vazia e `g G Q a` constrói a concatenação de todos `δ G q a` tais que `q ∈ Q` . A função `δ̂` aplica a função de transição sobre todos os estados da entrada e o termo atual da sequência, fazendo o mesmo recursivamente com os termos resultantes e o próximo símbolo. Todas essas definições recebem os decisores de igualdade do AFN, que suprimimos para simplificar.

A função de transição é responsável pela computação da entrada que o autômato recebe. Formalizamos a linguagem aceita pelo AFN `G` como

```
lang := fun G w => ∃ q,
  q ∈ (δ̂ G (start_states G) w) /\ q ∈ (accept_states G)
```

Logo, para verificar se uma palavra é aceita pelo AFN – se `lang G w` –, basta-nos executar a função de transição sobre os estados iniciais e essa dada palavra e, depois, verificar se um dos termos resultantes é estado final.

Outra definição importante relacionada à de transições tange aos caminhos. Um caminho é um meio de percorrer a estrutura do AFN, semelhante aos caminhos de grafos (BONDY; MURTY et al., 1976), em que começamos em um nó raiz e seguimos para outros nós a partir das arestas que partem do nó atual. Esta é uma definição indutiva de dois construtores, os quais determinam que:

- para todo possível estado[2] `q` , existe um caminho de `q` a `q` por `nil` ;
- para todos os estados `q1` , `q2` , símbolo `a` e palavra `w` , se existe uma transição `trans q2 a q3 ∈ G` e um caminho de `q1` a `q2` por `w` , então existe um caminho de `q1` a `q3` por `w ++ [a]` .

em que `++` é o operador de concatenação. Cada construtor recebe os termos e provas das hipóteses e produz uma prova para a existência de caminho – o presente trabalho

[2] Termo do tipo dos estados do autômato

nomeou esse tipo proposição como `path`. A noção de caminho é relevante para tratar de buscas[3] e da estrutura dos autômatos, como é apresentado adiante neste texto.

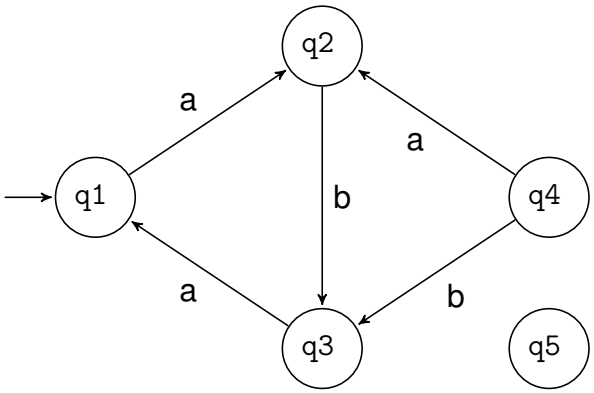
3.4 ALCANÇABILIDADE DOS ESTADOS

A noção de alcançabilidade relativa tange à definição de existência de caminho a partir de um dado estado de referência. Nesse sentido, quando existe caminho de `q` a `q'`, dizemos que o estado `q'` é alcançável a partir de `q`. De outro modo, os ditos simplesmente estados alcançáveis são aqueles para os quais existe algum caminho que leve de um estado inicial a eles. Equivalentemente, podemos definir a alcançabilidade de um estado `q` como

```
∃ w, q ∈ (δ̂ G (start_states G) w)
```

que vai ao encontro da noção com caminhos. A Figura 5 ilustra um AFN com dois estados inalcançáveis `q4` e `q5`, aos quais não existe palavra que transicione do estado inicial a eles.

Figura 5 – AFN com estados inalcançáveis



Fonte: Elaborada pelo autor, 2021.

Garantimos a decidibilidade da alcançabilidade de um estado mediante uma busca completa no AFN a partir dos estados iniciais como raízes, já que somente em caso positivo existe caminho de um estado inicial para o dado estado. A fim de provar o corretismo da busca para essa finalidade, mostramos que a busca de profundidade `n` percorre todos os caminhos de tamanho até `n` a partir dos estados iniciais. Depois provamos que todo estado alcançável é alcançado por uma palavra menor que o

[3] Como a busca em profundidade

tamanho do conjunto dos estados do autômato. Portanto, a busca com profundidade `|states G|` é completa.

Os estados inalcançáveis são inúteis para a computação do AFN e, removidos, não alteram a linguagem do autômato. Portanto, a minimização do AFN não deve conter tais estados.

3.5 AUTÔMATOS FINITOS DETERMINÍSTICOS

Autômatos finitos determinísticos (AFDs) carregam duas restrições a mais: (1) possuem apenas um estado inicial e (2) as transições são determinísticas, ou seja, não existem duas transições partindo de um mesmo estado e símbolo cujos estados destinos são diferentes. Consideremos permitida a existência de AFDs 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 não são impactadas. Para representar os AFDs no Coq, utilizemos a mesma estrutura de dados e um tipo proposição indutivamente definido. Basicamente, esse tipo (`dfa`) terá um construtor para cada construtor de AFN, com diferenças para os de estado inicial e transição. Nesses casos, adicionamos hipóteses para garantir as restrições. Cada construtor garante que:

- um AFN vazio `G = []` é determinístico;
- para quaisquer AFD `G:nfa A B` e termo `q:A` , temos o AFD `state q::G` : independentemente de qual seja `q` , ele pode ser concatenado como estado do autômato;
- para quaisquer AFD `G:nfa A B` e termo `a:B` , o AFN `symbol a::G` é determinístico: não importa se `a` é símbolo do autômato original, podemos adicioná-lo ao AFD;
- `accept q::G` é AFD se `G` também é: novos estados finais não interferem no determinismo;
- para qualquer AFD `G` , `start q::G` é AFD também se `q` é um estado inicial de `G` : como `q` já é um estado inicial, ele, duplicado, não interfere na multiplicidade

de estados iniciais distintos, o que queremos inibir;

- para qualquer AFD `G`, `start q::G` é AFD também se `G` não tem nenhum estado inicial: `q` será o único estado inicial do AFD, como requer a restrição;
- `trans q a q'::G` é um AFD se `G` também é e `trans q a q'` é uma transição original de `G`: a transição será duplicada, o que não gera transição não determinística;
- `trans q a q'::G` é um AFD se `G` também é e `G` não possui nenhuma transição `trans q a q''` (para qualquer `q''`): como δ `G q a = []`, o novo valor da função de transição aplicada sobre `q` e `a` será um conjunto unitário; ou seja, determinístico.

O fato de a nossa definição ser indutiva permite-nos inferir proposições sobre AFDs mediante provas por indução simples. Além disso, ao optarmos por construtores de fácil entendimento, conseguimos assegurar que nossas formulações estão corretas.

3.6 EQUIVALÊNCIA DE ESTADOS

Estados equivalentes, ou indistinguíveis, são aqueles que, para qualquer palavra `w`, a transição a partir deles aplicada em `w` resulta em estado final em ambos ou nenhum dos casos (HOPCROFT; MOTWANI; ULLMAN, 2006):

```
(∃ q1', (q1' ∈ (δ̂ G [q1] w) /\ q1' ∈ (accept_states G))) <->
  ∃ q2', (q2' ∈ (δ̂ G [q2] w) /\ q2' ∈ (accept_states G))
```

em que `q1` e `q2` são quaisquer estados equivalentes. Essa propriedade permite-nos substituir as ocorrências de `q1` ou `q2` pelo seu equivalente no autômato sem mudanças na linguagem. Prova disso é que, para quaisquer `Q` e `w1` tais que `q1 ∈ (δ̂ G Q w1)`, `w1 ++ w'` vai continuar sendo aceita ou rejeitada a partir de `Q` com a troca de `q1` por `q2`. Desse modo, concluímos que um AFD mínimo não pode ter estados equivalentes distintos; isto é,

```
∀ q1 q2, (q1 ≡ q2 -> q1 = q2)
```

o que o presente trabalho visa demonstrar para o algoritmo de Brzozowski.

A equivalência de estados pode ser generalizada a fim de valer para estados de AFNs diferentes:

```
(∃ q1', (q1' ∈ (δ̂ G1 [q1] w) /\ q1' ∈ (accept_states G1))) <->
  ∃ q2', (q2' ∈ (δ̂ G2 [q2] w) /\ q2' ∈ (accept_states G2))
```

que pode ser útil para descobrir se dois AFDs são equivalentes.

3.6.1 Equivalência de autômatos finitos determinísticos

Dois AFNs `G1` e `G2` são ditos equivalentes se

```
∀ w, (lang G1 w <-> lang G2 w)
```

ou seja, se ambos aceitam as mesmas palavras. Esse é um conceito fundamental para provarmos o corretismo dos algoritmos de minimização, uma vez que eles devem retornar um autômato equivalente ao original.

Podemos transferir os problemas de equivalência de AFDs com estado inicial para os da equivalência de estados, pois eles serão equivalentes se e somente se o estado inicial de um for equivalente ao do outro. A prova disso é simples, seja `w` qualquer palavra, ela será aceita ou rejeitada por ambos se e somente se

```
(∃ q1', (q1' ∈ (δ̂ G1 (start_states G1) w) /\ q1' ∈ (accept_states G1)))
<-> ∃ q2', (q2' ∈ (δ̂ G2 (start_states G2) w) /\ q2' ∈ (accept_states G2))
```

Como esses autômatos são determinísticos e tem estado inicial,

```
(∃ q1', (q1' ∈ (δ̂ G1 [q1] w) /\ q1' ∈ (accept_states G1))) <->
  ∃ q2', (q2' ∈ (δ̂ G2 [q2] w) /\ q2' ∈ (accept_states G2))
```

em que `q1` e `q2` são os respectivos estados iniciais. Um raciocínio semelhante pode ser aplicado para os caminhos.

Se ambos não possuem estados inciais, ambos são obviamente equivalentes: não são capazes de aceitar uma sequer palavra. Já no caso de apenas um deles ter estado inicial, a decidibilidade de serem equivalentes reduziremos ao problema de verificar se algum estado final daquele que tem estado inicial é alcançável. Em positivo,

concluímos que não são equivalentes, pois existe palavra que é aceita em um, e o outro rejeita todas. Por outro lado, em negativo, sabemos que são equivalentes pois a linguagem de ambos é vazia.

A equivalência de AFDs é necessária para garantirmos que a minimização completa não altera a linguagem de um autômato. Por isso, o presente trabalho desenvolveu provas inspiradas na demonstração desta seção.

3.7 REVERSÃO

A reversão de uma palavra resulta no espelhamento dos símbolos da palavra no que tange à ordem deles. O reverso de `[1;1;0;1]`, por exemplo, é `[1;0;1;1]`; o de `[0;1;1;0]` é a própria `[0;1;1;0]` – configurando um palíndromo, palavra igual a seu reverso (FERREIRA, 2018). Um algoritmo de reversão de palavras é uma recursão simples:

```
Fixpoint w^R :=
  match w with
  | nil => nil
  | a::w => (w^R) ++ [a]
  end
```

Já a reversão de um AFN é um processo que inverte a ordem das transições: a origem vira destino e vice-versa. Além disso, é parte dessa transformação a inversão de qualidade dos estados; ou seja, os iniciais viram finais e vice-versa. Programando temos

```
Fixpoint rev G :=
  match G with
  | nil => nil
  | trans q a q'::G' => trans q' a q::G'
  | start q::G' => accept q::rev G'
  | accept q::G' => start q::rev G'
  | x::G' => x::rev G'
  end
```

A linguagem de um AFN `G` revertido deve ser reversa: todas as palavras da linguagem original revertidas. É fácil perceber isso, pois, com a reversão, todos os caminhos são revertidos, tal como a qualidade dos estados.

A reversão de autômatos é importante para o algoritmo de Brzozowski, motivação do presente estudo. Outro passo importante nesse algoritmo é a determinização, de que trata a seção seguinte.

3.8 DETERMINIZAÇÃO

Apesar do alto desempenho, os AFNs estritos[4] não podem ser implementados mediante paralelismo, já que não existe máquina capaz de realizar computações em qualquer número de estados simultâneos. O que podemos fazer é simular o comportamento de um AFN em uma espécie de busca em largura. Na prática, isso equivale a uma determinização do autômato, na qual os novos estados `Q` são conjuntos de estados tais que, aplicada a nova função de transição $\hat{\delta}$ `(det G)` a `Q`, resulta na função de transição original $\hat{\delta}$ `G` aplicada ao conjunto de estados que ele representa:

```
∀ q, (q ∈ (δ̂ (det G) [Q] w) <-> q ∈ (δ̂ G Q w))
```

O único estado inicial do AFN determinizado é o conjunto dos estados iniciais original, porque é a partir de todos esses estados que se inicia a computação das entradas pelo AFN. Já para construirmos o conjunto dos estados finais, precisamos determinar a maneira de obter as transições e os estados. Uma maneira mais intuitiva, mas menos otimizada, é por usar o conjunto das partes dos estados: todos os possíveis subconjuntos. Por outro lado, é provável que alguns ou muitos desses estados sejam inalcançáveis e, por conseguinte, desnecessários. Nesse sentido, com a construção do AFD por busca a partir dos estados inciais como nó raiz, temos mais eficiência – construímos somente os estados alcançáveis.

A demonstração do corretismo do algoritmo de determinização que este trabalho desenvolveu, bem como de outros lemas importantes relativos a esse processo, são aprofundados na seção 6.3. Esse algoritmo faz parte da minimização de Brzozowski, razão por que sua introdução é imprescindível para este estudo.

[4] Desconsiderando os AFDs, que também são AFNs

3.9 APLICAÇÕES

As expressões regulares são muito empregadas para verificar padrões de textos e códigos. Exemplos dessas aplicações incluem sequenciamento genético (MARASS; UPTON, 2009), detecção de intrusão (VASILIADIS et al., 2009), mineração de dados (SAVCHENKO, 2002) e validação de formulários. Como toda expressão regular especifica uma linguagem regular, os AFNs são muito importantes nesse contexto. A verificação de casamento de *string* com expressão regular é feita mediante AFDs implementados nas diversas linguagens de programação existentes.

Além das expressões regulares, os AFNs são utilizados para modelar sistemas a eventos discretos (SEDs). Segundo Lafortune (2019, p. 142, tradução do autor)

> Sistemas a eventos discretos (SEDs) são sistemas dinâmicos com duas características definidoras: seus espaços de estados são discretos e potencialmente infinitos, e sua dinâmica é orientada a eventos, em vez de tempo. (...) Eventos que ocorrem de forma assíncrona (em geral) causam um salto no espaço de estados, de um estado para outro.

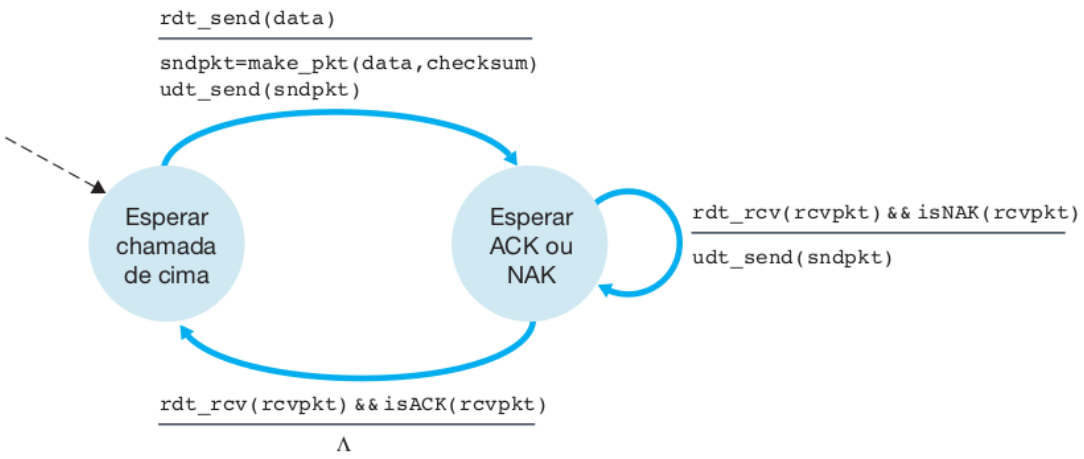
Cassandras e Lafortune (2008) argumentam que os SEDs, diferentemente dos sistemas orientados pelo tempo e governados pelas leis da natureza, não podem ser modelados e estudados usando equações diferenciais. Os SEDs requerem outras técnicas de análise, ferramentas de projeto, métodos de teste, entre outros.

O advento das tecnologias de comunicação, sensoriamento e computação impulsionou a caracterização desta nova classe de sistemas. Os SEDs, por não serem adequadamente modelados pelas equações das leis da física clássica e moderna, demandam novos paradigmas de modelagem. “Intuitivamente, nós podemos pensar em um modelo como um dispositivo que simplesmente duplica o comportamento do próprio sistema” (CASSANDRAS; LAFORTUNE, 2008, p. 3, tradução do autor). Um modelo descreve um sistema sob uma ótica quantitativa, que é, muitas vezes, mais adequada que uma simples e subjetiva descrição qualitativa. A construção de um modelo considera um conjunto de variáveis de entrada e saída, relacionadas por intermédio de uma função. Assim sendo, os modelos proporcionam a habilidade predizer os comportamentos de sistemas, o que os confere importância.

Cassandras e Lafortune (2008) citam, como exemplos de SEDs, os sistemas de filas, computacionais, de comunicação, de manufatura e de tráfego. A exemplo

disso, protocolos de comunicação em redes de computadores são frequentemente modelados por SEDs, como ilustra a Figura 6. O diagrama dessa figura representa um AFD com saída, mais especificamente uma máquina de Mealy, sendo a saída associada às transições (AARTS; VAANDRAGER, 2010).

Figura 6 – Protocolo para um canal de redes de computadores



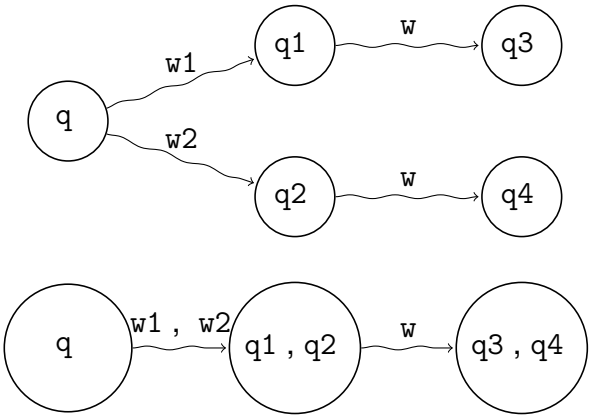
Fonte: KUROSE, J.; ROSS K. **Redes de computadores e a internet**: uma abordagem top-down. 6. ed. São Paulo: Pearson, 2013.

Além das aplicações supracitadas, os AFNs são base para a análise léxica dos compiladores, responsável por quebrar o programa de entrada em tokens a serem processados pelos demais processos da compilação desse programa. Tais autômatos reconhecem os padrões especificados para cada token, o que permite a classificação dos lexemas do programa (HOPCROFT; MOTWANI; ULLMAN, 2006).

4 MINIMIZAÇÃO DE AUTÔMATOS FINITOS DETERMINÍSTICOS

Suponhamos que um determinado AFD tenha dois estados `q1` e `q2` equivalentes e alcançáveis a partir de um outro estado `q` por `w1` e `w2` respectivamente. Quaisquer dois caminhos que comecem em `q` e percorram `w1` e `w2` são equivalentes no sentido de que `w1 ++ w` e `w2 ++ w` são ambas aceitas ou rejeitadas. Portanto, em termos da linguagem do autômato, há uma duplicação desnecessária de caminhos. A Figura 7 ilustra uma aglutinação com possíveis ganhos: menor complexidade de espaço, além de simplificar o entendimento de como o AFD funciona. A complexidade de espaço fica menor porque menos estados são armazenados, o que também pode tornar a computação prática mais rápida devido ao enxugamento da estrutura de dados que implementa o autômato.

Figura 7 – Aglutinação de caminhos



Fonte: Elaborada pelo autor, 2021.

Em suma, o AFD mínimo realiza as mesmas tarefas que o autômato original, com o diferencial de ter menos estados e transições distintos. Hajam vista os benefícios dessa simplificação, as seções a seguir discutem duas técnicas de minimização consolidadas.

4.1 MINIMIZAÇÃO PELA UNIÃO DOS ESTADOS EQUIVALENTES

O método de Hopcroft, Motwani e Ullman (2006) faz a minimização do AFD `G` por meio da verificação de equivalência de todos os pares de estados do AFD. Depois, ele particiona os estados, mantendo em blocos os equivalentes entre si, e constrói as

novas transições conforme a Figura 7.

Como forma de construir os novos estados e transições resultantes deste algoritmo, podemos pôr todos os estados originais em blocos unitários, remover o estado $\{q2\}$ do autômato e renomear o estado equivalente $\{q1\}$ para incluir o estado `q2` como $\{q1, q2\}$. Em seguida, fazemos o mesmo com os demais estados equivalentes a `q2` e `q1`, adicionando-os ao bloco que agora é $\{q1, q2\}$. Observemos a propriedade transitiva dessa relação: se `q1` é equivalente a `q2` e `q2` é equivalente a algum `q'`, então `q1` é equivalente a `q'`. Isso nos garante que cada estado só pode estar em um bloco. Além desta, a propriedade comutativa é clara, e a demonstração para ambas é útil não apenas para a apuração deste algoritmo, mas da técnica de Brzozowski.

Ainda no algoritmo de Hopcroft, Motwani e Ullman (2006), o novo estado inicial será o bloco com o estado inicial original, e os novos estados finais serão os blocos que têm algum estado final – todos os estados desses blocos serão finais, pois estados finais só podem ser equivalentes a estados finais. A minimização é garantida porque supomos que o AFD original não possui estados inalcançáveis. Não obstante, podemos aplicar algum decisor de alcançabilidade e remover tais estados quando houver (ou fazer a remoção manual); isso permite aplicar o algoritmo para qualquer AFD. Essa mesma regra vale para o próximo algoritmo. O escopo do presente trabalho, entretanto, é restrito aos AFDs sem estados inalcançáveis.

4.2 ALGORITMO DE BRZOZOWSKI

Diferente do algoritmo anterior, o algoritmo de Brzozowski e Tamm (2014) não requer a verificação dos pares de estados equivalentes. Fazemos a construção de um AFD mínimo com linguagem reversa por meio da reversão do AFN de entrada `G` e, em seguida, da determinização do retorno dessa transformação:

```
brzozowski := fun G => det (rev G)
```

O que queremos provar neste trabalho é que, se `G` é um AFD sem estados inalcançáveis, `brzozowski G` é um AFD mínimo que reconhece as palavras reversas às quais o autômato `G` aceita. O resultado dessas operações permite-nos encontrar o AFD mínimo equivalente a `G` aplicando novamente o algoritmo:

```
brzozowski_complete := fun G => brzozowski (brzozowski G)
```

de modo que `brzozowski_complete G` é mínimo e tem a mesma linguagem de `(G^R)^R = G`, admitido o corretismo da função `brzozowski`.

Em comparação com o algoritmo de Hopcroft, Motwani e Ullman (2006), o de Brzozowski tem um funcionamento mais simples e, por isso, pode ser mais fácil de compreender e aplicar. Além disso, apesar de ter complexidade maior, o algoritmo costuma ser eficiente na prática. De acordo com Tabakov e Vardi (2005), em se tratando de tempo de execução, ele é melhor em alta densidade de transições[1] que o método da Seção 4.1.

[1] Razão entre o número de transições e o número de estados

5 TRABALHOS RELACIONADOS

Doczkal, Kaiser e Smolka (2013) apresentam uma teoria construtiva de linguagens regulares em Coq na qual provam o teorema de Kleene, o lema do bombeamento, a unicidade do autômato determinístico mínimo, o teorema de Myhill-Nerode e propriedades sobre expressões regulares. No trabalho, os autores definem as expressões regulares de maneira indutiva no Coq e formalizam AFDs e AFNs com tipos *record*, estruturas dotadas de atributos. Para garantir que os tipos dos estados e símbolos dos autômatos sejam finitos, eles utilizam uma extensão do Coq e o tipo `char` respectivamente. O trabalho resultou em cerca de 1400 linhas de código e 170 lemas.

Outro trabalho relacionado que implementa autômatos usando tipos *record* do Coq é o de Athalye (2017), que versa sobre autômatos de entrada e saída: *IO automata*. Tais autômatos são, como o autor define, modelos de componentes de sistemas distribuídos, servindo de formalismos matemáticos para garantir o corretismo de projetos e implementações de algoritmos distribuídos e protocolos. O trabalho modela SEDs por meio de autômatos e prova características sobre esses modelos no Coq. Como motivação do estudo, o autor destaca a explosão combinatorial decorrente do *model checking*, o que faz essa técnica ser inutilizável na verificação de sistemas reais.

Braibant e Pous (2011) propõem uma tática para resolver equações e inequações em álgebras de Kleene, uma porção de relações binárias decidíveis não triviais constituída pelas constantes e operadores das linguagens regulares. Para verificar se duas expressões regulares expressam a mesma linguagem, os autores desenvolveram um algoritmo que constrói um AFN com transições vazias para cada expressão, converte-os para AFDs e verifica a equivalência entre eles. O desenvolvimento foi feito no Coq, bem como a prova de que ele é correto. O trabalho originou um algoritmo de 10 mil linhas e eficiente, se comparado seu desempenho de tempo com o dos outros algoritmos.

A prova da minimização para o algoritmo de Brzozowski de Bonchi et al. (2012) utiliza álgebra e coálgebra para modelar respectivamente a alcançabilidade e observabilidade, propriedade da inexistência de pares de estados indistinguíveis, e tem como base a dualidade entre elas. Os autores estendem a aplicabilidade do algoritmo a Máquinas de Moore determinísticas, que são uma espécie de AFD com saída. Apesar de

ser um trabalho correlato, as técnicas de demonstração que os autores empregaram é distante da que baseia o presente trabalho, que busca técnicas mais presentes no cotidiano do estudante de ciência da computação. Como proveito, tiramos do estudo a condição de que um AFD é mínimo se ele é observável; em outras palavras, se não tem estados inalcançáveis.

O trabalho relacionado executado por Paulson (2015) apresenta uma forma de representar os AFNs no assistente Isabelle por meio de conjuntos hereditariamente finitos, cuja construção é indutiva de duas formas:

- caso base: $\varnothing$;
- caso recursivo: dados quaisquer conjuntos hereditariamente finitos, sua união também será um conjunto hereditariamente finito.

e cuja representação podemos efetuar por meio de listas, funções ou naturais. A técnica do autor utiliza um comando do Isabelle semelhante e homônimo aos *records* do Coq, uma abordagem outrora tentada pelo presente trabalho, em que são definidas as condicionais para as restrições dos conjuntos componentes dos AFNs. Paulson (2015) também versa sobre a minimização e, nesse contexto, define que um AFD é mínimo se todos os estados forem alcançáveis e distinguíveis – mesma solução que seguimos. Outro aspecto que distingue esse trabalho e o presente estudo tange à construção do algoritmo de Brzozowski: primeiramente o autor reverte o AFD de entrada, determiniza o AFN gerado por meio da explosão de estados e, por fim, remove os estados inalcançáveis. Como já suscitamos anteriormente, essa técnica de construção de AFD pelo conjunto das partes não é ótima e requer esse passo a mais no algoritmo. Para tanto, também mudam as provas.

Os trabalhos relacionados enumerados por este capítulo inspiraram o desenvolvimento das definições que inserimos e das demonstrações que seguem. O presente trabalho também tem fundamento em informações da internet, que objetivamos demonstrar.

6 ALGORITMO DE BRZOZOWSKI É CORRETO

Como já inserido na Seção 4.2, o algoritmo de Brzozowski e Tamm (2014) pode ser escrito como

```
brzozowski := fun G => det (rev G)
```

Para obtermos o AFD mínimo para a linguagem original, temos de executar duas vezes o algoritmo, já que a linguagem do autômato resultante é reversa. Na realidade, a primeira execução do programa já retorna um autômato que é mínimo; por isso, a fim de demonstrar seu corretismo à luz de sua finalidade, basta-nos provar que o resultado sempre é um AFN determinístico e mínimo e decide a linguagem reversa. O presente trabalho desenvolveu as provas que seguem na linguagem do assistente de provas Coq, com a validação do mesmo.

O fato de podermos descrever o algoritmo em uma única linha mostra que seu funcionamento é simples por si só, mas ele depende dos algoritmos de reversão (`rev`) e determinização (`det`). Inicialmente, consideremos esses programas como caixas pretas funcionando corretamente. Nosso objetivo é demonstrar que, para qualquer autômato finito determinístico `G` sem estados inalcançáveis:

1. `brzozowski G` é um autômato finito determinístico (Subseção 6.3.4);
2. `lang (brzozowski G)` é a linguagem reversa de `G` (Seção 6.2 e Subseção 6.3.5);
3. todos os estados de `brzozowski G` são alcançáveis (Subseção 6.3.3);
4. todos os pares de estados de `brzozowski G` são distinguíveis, ou não equivalentes.

Os itens (3) e (4), conforme explicam Bogdanov (2010), Bonchi et al. (2012), são suficientes para mostrar que o autômato resultante é minímo.

À primeira vista, `brzozowski G` é garantidamente um AFD se

```
∀ A B (G:nfa A B), dfa (det G)
```

Cabe ressaltar que, entretanto, como veremos, essa proposição é condicionada aos decisores de igualdade de `A` e `B`. Na prática, definimos o algoritmo de Brzozowski de maneira um pouco mais complexa:

```
brzozowski := fun A B eqA eqB (G:nfa A B) => det eqA eqB (rev G)
```

função cujo tipo é

```
∀ A B, ((A->A->bool)->(B->B->bool)->(nfa A B)->(nfa A B))
```

que é um tipo dependente, pois os decisores e o próprio AFN dependem dos dois primeiros termos da entrada. Ao observar o tipo da função, podemos duvidar a princípio de algum dos tipos colocados. Ao invés, não seria correto o tipo que segue?

```
∀ A B, ((A->A->bool)->(B->B->bool)->(dfa A B)->(dfa A B))
```

Lembremos que `dfa` não é um tipo de estrutura de dados, mas um uma função proposicional indutivamente definida que engloba as restrições dos AFDs. Logo, precisamos separar: `nfa` é o que define a estrutura de dados dos AFNs em geral, e `dfa` é uma função proposicional sobre a qualidade determinística de um dado AFN representado por uma lista de algum tipo `nfa A B`. Logo, com

```
∀ eqA eqB A B (G:nfa A B), (eq_dec eqA -> eq_dec eqB ->
    dfa eqA eqB (det G))
```

reescrevemos, de forma completa, o lema que queremos provar sobre o resultado da função `det` ser um AFD de fato. A proposição `eq_dec eq` indica que a função `eq` decide a igualdade do tipo das suas entradas. As hipóteses, pois, servem para assegurar que as funções `eqA` e `eqB` são, de fato, decisores para a igualdade de `A` e `B` respectivamente e são necessárias porque precisamos fazer comparações durante a busca que executamos sobre o autômato de entrada. No entanto, como estamos supondo que as entradas de nossas funções são AFNs corretamente especificados, a existência dos respectivos decisores é necessariamente verdadeira; então, podemos omiti-los. Portanto, o item (1) do nosso objetivo está espontaneamente provado se provarmos o mesmo para o algoritmo `det`.

Em se tratando da linguagem do autômato resultante, outrossim, reduziremos o item (2) aos demais algoritmos. Em primeiro lugar, a linguagem de `rev G` deve

ser a reversa da de `G`. Além disso, `det G'` deve manter a linguagem de qualquer `G'` intacta. O AFN retornado por `brzozowski`, pois, aceitará a linguagem reversa do autômato de entrada, como queríamos demonstrar. Nesse mesmo sentido, mais uma vez, se todos os estados do AFN retornado pelo algoritmo de determinização forem alcançáveis, claramente, `brzozowski G` só terá estados alcançáveis.

Um pouco menos trivial, a afirmação do item (4) subdividiremos em dois lemas. Vale realçar que a inspiração para as provas desses lemas vem de uma publicação em um fórum de ciência da computação de internet. Jan (2019) apresenta uma forma aparentemente trivial de alcançar esse objetivo. O primeiro lema que ele define, a "observação básica", é uma relação de equivalência entre as funções de transição do autômato original e do minimizado. Em seguida, formaliza uma prova para

```
∀ q1 q2, (q1 ∈ states (brzozowski G) -> q2 ∈ states (brzozowski G) ->
    q1 ≡ q2 -> q1 = q2)
```

Embora a prova do autor esteja incompleta, ela mostra que demonstrar o corretismo do algoritmo de Brzozowski pode ser um pouco mais simples que a forma exposta por outros trabalhos relacionados. Por esse motivo, o presente estudo visa enfatizar essa constatação.

Antes de inserirmos a observação básica sobre o algoritmo, precisamos definir uma função parcial de transição específica para os AFNs que são determinísticos. Pelo fato de o Coq apenas permitir funções totais, precisamos torná-la total. A definição que segue faz isso

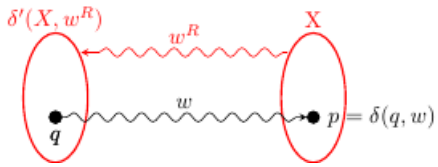
```
δ* := fun G q w =>
    match δ̂ G q w with
    | nil => None
    | q'::_ => Some q'
    end
```

em que `None` indica o indefinido, e `Some`, um valor definido. Sabemos que a transição de um AFD é sempre determinística; ou seja, ela será vazia ou indicará apenas um possível caminho. Por essa razão temos garantida a propriedade

```
∀ q w, (dfa G ->
    ∀ q', (q' ∈ (δ̂ G q w) -> δ* G q w = q'))
```

cuja prova nós desenvolvemos e validamos no assistente Coq com esse mesmo raciocínio.

Figura 8 – Observação básica sobre o algoritmo de Brzozowski



Fonte: Jan (2019).

A observação básica de Jan (2019) trata da relação entre as transições estendidas do AFD de origem e o retornado pelo algoritmo de Brzozowski. Ela especifica que um estado `q` de `G` pertence ao estado da transição partindo de algum outro estado `Q` do autômato resultante `brzozowski G` se e somente se a mesma palavra percorrida, revertida, ocasiona a transição de `q` a um estado original que pertence a `Q`:

```
∀ q Q w, (Q ∈ states (brzozowski G) ->
    (q ∈ (δ* (brzozowski G) Q w) <-> (δ* G q (w^R)) ∈ Q))
```

sem necessidade de condicionar `q` como estado de `G`, pois, ao admitirmos que o algoritmo de determinização está corretamente construído nos moldes da Seção 3.8,

```
∀ q Q, (Q ∈ states (brzozowski G) -> q ∈ Q -> q ∈ (states G))
```

é necessariamente verdadeira.

Já falamos sobre a equivalência de transição e caminho, razão por que temos a oportunidade de utilizar a noção de caminhos para chegar à conclusão do lema dessa observação. Reescrevamos:

```
(∀ Q', (q ∈ Q' -> path (brzozowski G) Q Q' w -> ∃ p, (p ∈ Q /\
    path G q p (w^R)))) /\
∀ p, (p ∈ Q -> path G q p (w^R) -> ∃ Q', (q ∈ Q' /\
    path (brzozowski G) Q Q' w))
```

a partir de tal equivalência. Com base na reversão de autômatos, façamos a substituição de `G` por `rev G = G'` no lema obtido para encontrarmos:

```
(∀ Q', (q ∈ Q' -> path (det G') Q Q' w -> ∃ p, (p ∈ Q /\ path G' p q w)))
/\
∀ p, (p ∈ Q -> path G' p q w -> ∃ Q', (q ∈ Q' /\ path (det G') Q Q' w))
```

representada pela inversão da transição em preto da Figura 8. Generalizando o termo `G'` como um AFN com um estado final, o nosso trabalho de demonstrar a observação ficará no escopo específico do algoritmo de determinização. Isso será discutido posteriormente neste texto, mas, admitindo ser verdadeira, temos provada a observação básica.

Mediante a observação básica e sucessivas reescritas, podemos adiante demonstrar o último item de nosso objetivo. Queremos provar que dois estados de `brzozowski G` serão equivalentes apenas se forem iguais. Com isso, teremos que não há estados indistinguíveis no AFD resultante.

Sejam `Q1` e `Q2` estados indistinguíveis de `brzozowski G`. Por definição,

```
∀ w, ((δ* (brzozowski G) Q1 w) ∈ accept_states (brzozowski G) <->
    (δ* (brzozowski G) Q2 w) ∈ accept_states (brzozowski G))
```

Pelo funcionamento da determinização, um estado deve ser final se e somente se a ele pertence algum estado final do autômato original – neste caso, o estado final do algoritmo de reversão. Pela construção da reversão de AFN, os estados finais devem trocar de qualidade com os iniciais. Assim, o estado final do AFD revertido é `q0 ∈ start_states (G)` ou `nil`, porque um AFD pode ter no máximo um estado inicial. Então, podemos separar a nossa análise em dois casos.

Se o AFD original não tiver estado inicial, a linguagem será vazia, bem como esta revertida. O presente trabalho, em conformidade com a Seção 3.8, desenvolveu um algoritmo de determinização que gera um autômato vazio quando não houver estado final. No caso do autômato não ter estados iniciais, o revertido não terá estados finais. Assim, o algoritmo determinizador verificará que `rev G` não possui estados finais e retornará `nil`. Portanto, nesse sentido, o AFD `det (rev G)` não terá estados indistinguíveis – aliás, não terá estado algum.

Embora não tenha sido a abordagem utilizada pelo trabalho, poderíamos ter feito uma outra prova por contradição no lugar da exposta. Estamos supondo que o

AFD `G` não tem estados inalcançáveis; com isso, precisa existir caminho dos estados iniciais para os estados `Q1` e `Q2`, um absurdo.

Já no caso de `G` ter o estado inicial `q0`, extraímos

```
∀ w, (q0 ∈ (δ* (brzozowski G) Q1 w) <-> q0 ∈ (δ* (brzozowski G) Q2 w))
```

já que a função `det`, como veremos, marca como finais os estados aos quais pertence um estado final do AFN de entrada (`q0`). Ao reescrevermos a observação básica na equação, obteremos

```
∀ w, ((δ* G q0 w) ∈ Q1 <-> (δ* G q0 w) ∈ Q2)
```

o que é suficiente para mostrar que `Q1 = Q2`, pois, em outras palavras, isso quer dizer que todo estado alcançável `q ∈ states G` pertencente a `Q1` se e somente se pertence a `Q2`. Como todos os estados em `states G` são alcançáveis, todos os estados pertencentes a `Q1` pertencem a `Q2` e vice-versa. Assim, `Q1` é um conjunto equivalente ao `Q2`. Mais que isso, pelo fato de nossa determinização normalizar os conjuntos equivalentes – ou seja, fazer a substituição –, `Q1 = Q2` como queríamos demonstrar.

Dado que a base da prova do corretismo do algoritmo desenhado por Brzozowski e Tamm (2014) está exposta, falta-nos provar os lemas de que aqui lançamos mão. Comecemos pela simples reversão de autômatos e posteriormente provemos, baseado nos artefatos que este trabalho desenvolveu no assistente Coq, as proposições relacionadas ao processo de determinização.

6.1 REVERSÃO DA QUALIDADE DOS ESTADOS

Explicitamos neste capítulo que é fundamental termos como verdadeira a proposição que inverte a qualidade dos estados no processo de reversão. O algoritmo de reversão de AFNs e palavras implementados no Coq são os que especificamos na Seção 3.7. A prova que faremos acerca dos estados finais e iniciais do resultado de `rev G` é mediante indução em `G`. A base, com `G = nil`, demonstramos por contradição, haja vista a total ausência de estados finais ou iniciais. No passo indutivo podemos dividir o lema. Inicialmente temos

```
q ∈ (accept_states (c::G))
```

e a hipótese indutiva

```
q ∈ (accept_states G) -> q ∈ (start_states (rev G))
```

Observamos duas possibilidades: (1) `c = accept q` ou (2) `q ∈ (accept_states G)` . Do caso (1) tiramos

```
q ∈ (start_states (start q::rev G))
```

Logo

```
q ∈ (start_states (rev (c::G)))
```

como desejávamos demonstrar. Aplicada a hipótese indutiva no segundo caso, obtemos

```
q ∈ (start_states (rev G))
```

Outrossim, independente de qual seja `c` ,

```
q ∈ (start_states (rev (c::G)))
```

De maneira semelhante obtemos a prova de

```
∀ q, (q ∈ (start_states G) -> q ∈ (accept_states (rev G)))
```

Podemos questionar o porquê de apenas provarmos um lado das bi-implicações. Não precisamos provar a recíproca; afinal, `G = rev (rev G)` . A demonstração dessa igualdade também efetuamos por indução. A base, `nil = rev (rev nil)` , é trivial. O passo é

```
c::G = rev (rev (c::G)) = rev (rev (c::nil)) ++ (rev (rev G))
```

Reescrevemos a hipótese de indução `G = rev (rev G)`

```
c::G = rev (rev (c::nil)) ++ G
```

e com análise de casos para `c` , encontramos

```
c::G = c::nil ++ G
```

terminando a prova. Essa igualdade permiti-nos afirmar o contrário da reversão da qualidade dos estados. Por exemplo,

```
∀ q, (q ∈ (start_states (rev G)) -> q ∈ (accept_states G))
```

já que, substituindo `rev G` por `G'` e usando a igualdade que acabamos de provar,

```
∀ q, (q ∈ (start_states G') -> q ∈ (accept_states (rev G')))
```

como já mostramos.

6.2 LINGUAGEM REVERTIDA É REVERSA

Sabemos que, para todo `w`, `lang G w` se e somente se `path G q0 qf w` com algum `q0 ∈ (start_states G)` e algum `qf ∈ (accept_states G)`. Se provarmos `path (rev G) qf q0 w`, então teremos como demonstrar que a linguagem do AFN revertido é reversa:

```
lang (rev G) w <-> lang G (w^R)
```

porque `q0 ∈ (accept_states G)` e `qf ∈ (start_states G)`, conforme observamos na Seção 6.1.

Por nossa definição de existência de caminho ser indutiva, podemos, como o nome sugere, fazer indução sobre a hipótese `path G q0 qf w`. A base é trivial, pois um caminho vazio sempre existe, e o passo engloba

```
trans q a qf ∈ G
```

e a hipótese indutiva

```
path (rev G) q q0 (w^R)
```

Garantimos

```
trans qf a q ∈ (rev G)
```

Portanto,

```
path (rev G) qf q0 (a::(w^R))
```

ou

```
path (rev G) qf q0 ((w ++ [a])^R)
```

como objetivávamos.

6.3 ALGORITMO DE DETERMINIZAÇÃO

Já discutimos anteriormente sobre como escrever um algoritmo de determinização por meio de uma espécie de busca no grafo do AFN de entrada, uma opção melhor que gerar uma explosão de estados mediante a construção de todos os possíveis estados do conjunto das partes. Consoante ao exposto, precisamos definir uma metodologia para o nosso programa. Ressaltemos o fato de que representamos conjuntos como listas e, dessarte, dois conjuntos equivalentes podem não ser iguais, tomando essa representação. No entanto, precisamos garantir que todos os estados resultantes da determinização sejam, quando equivalentes, idênticos na forma.

Neste contexto, uma abordagem que empregamos tange à normalização. Ela tem o papel de substituir todos os conjuntos equivalentes entre si por um destes. Assim, asseguramos que o AFN final tem a propriedade que desejamos:

```
∀ Q1 Q2, (Q1 ≡ Q2 -> Q1 = Q2)
```

Antes de aplicarmos a função normalizadora, no entanto, temos de definir quais serão os estados iniciais, finais, alfabeto e transições, a desconsiderar a distinção entre equivalência e igualdade de “conjuntos”. O estado inicial, já mencionado na Seção 3.8, é o conjunto dos estados iniciais do autômato original:

```
start_states (det G) = [start_states G]
```

O alfabeto deixaremos que sejam determinados implicitamente pelas transições. Estas serão o passo mais complexo – em tempo, espaço e compreensão – e ocorrerão pela busca supracitada. Os estados finais dependem dos demais para serem definidos. Seja a função

```
Fixpoint det_accept G L :=
    match L with
    | nil => nil
```

```
    | Q::L' =>
        if has_accept_state G Q then
            accept Q::det_accept G L'
        else
            det_accept G L'
    end
```

em que `has_accept_state G` é um decisor para a existência de estado final do autômato `G` na entrada, o qual utiliza os decisores de igualdade dele. Essa definição será útil para a montagem final do algoritmo, pois será responsável por marcar os estados finais, aqueles que possuírem algum estado final original.

Para a construção das transições da determinização, devemos definir antes uma função que servirá de `foreach` para o nosso programa funcional. Um `foreach` é um laço de programação no qual sucessivamente executamos iterações sobre cada um dos elementos de uma dada entrada (SETZER, 2002). No presente trabalho, o `foreach` receberá uma função `f` que executará cada iteração, um conjunto de estados `Q` e um conjunto de símbolos `L`. Ele será responsável por gerar as transições simples[1] a partir de `Q` – conjunto que será estado do autômato determinizado –, por meio da aplicação da função de transição sobre `Q` e os símbolos em `L`, e executar a função `f` recursivamente nos estados vizinhos gerados. Estados vizinhos são aqueles para os quais existe transição chegando a eles.

Agora só precisamos executar o laço recursivamente. Para tanto, precisamos definir mais uma função:

```
Fixpoint det_trans G fuel Q :=
    match fuel with
    | O => nil
    | S fuel' => foreach G (det_trans G fuel') Q (alphabet G)
    end
```

que fornecerá, recursivamente, as transições do AFN determinizado a partir da raiz da busca `Q`. Neste contexto, os números naturais são construídos indutivamente: zero e sucessor de um natural `n` (`S n`). À primeira vista parece estranho um número (`fuel`)

[1] Sem serem estendidas

limitando o alcance da função, mas ele faz sentido para garantir a parada do programa. Designamos a ele o nome de combustível, dado seu papel exclusivo de manter o algoritmo operando. Para o programa estar correto, deve haver um número limitante (combustível) que seja suficiente para computarmos todos os novos estados. Como a profundidade máxima da busca é o número máximo de novos estados possíveis, se considerarmos a relação de equivalência para distinguir um conjunto do outro, teremos até `2^|states G|` conjuntos estados diferentes (DEVLIN, 1979). Um valor maior ou igual a este será combustível suficiente.

Com o `det_trans` conseguimos, enfim, determinizar as transições do AFN. A normalização do estado inicial e as transições geradas será

```
h := fun G => normalize (start (start_states G)::
    det_trans G (2^|states G|) (start_states G))
```

Juntando todas as funções desta seção:

```
det := fun G =>
    match accept_states G with
    | nil => nil
    | _ => h G ++ (det_accept G (states (h G)))
    end
```

que é o nosso algoritmo de determinização completo, já com a marcação dos estados normalizados. Como mencionamos, ele retorna o vazio sempre que o AFN de entrada não tiver estados finais. Acerca da determinização, as próximas subseções visam demonstrar os lemas pertinentes ao algoritmo de Brzozowski que aproveitamos no início do capítulo.

6.3.1 Normalização

É essencial que o autômato gerado pelo algoritmo de determinização tenha a propriedade de equivalência-igualdade. É por tal razão que aplicamos uma normalização `normalize` ao fim da construção das transições do AFD. Ela é uma função que substitui todos os conjuntos equivalentes por um deles – no presente trabalho, o primeiro que aparece. Para tanto, a função, com o decisor da igualdade do tipo `A` dos

estados originais, consegue verificar, em uma lista de conjuntos representados pelo tipo `list A`, da esquerda para a direita, se o conjunto atual dessa lista é equivalente a outro da mesma lista. O presente trabalho demonstrou a garantia da propriedade após a normalização, mediante indução na entrada.

Nas próximas subseções, nós nos valeremos da manutenção de certas propriedades após a normalização. Isso torna possível fazer provas indutivas acerca do AFN anterior a ela de modo que as mesmas proposições valham para o autômato normalizado.

6.3.2 Estados finais

Uma proposição necessária e que assumimos anteriormente concerne aos estados finais da determinização. O conjunto dos estados finais de `det G` é o conjunto de todos os estados aos quais pertence algum estado final do autômato original `G`. Nossa prova começa destrinchando a definição `det`. Em primeiro lugar, se `G` não tem estados finais, o resultado é automaticamente vazio; logo, a proposição está conforme. Senão, precisamos destrinchar a concatenação `h G ++ (det_accept G (states (h G)))`. Na esquerda não há estados finais, porque essa lista só possui termos dos construtores `start` e `trans`, que a normalização mantém intactos. Para o lado direito da concatenação, `det_accept` marca todos os estados da sua entrada que tenham um estado final do autômato `G`. Esses estados são resultado da construção do algoritmo `det_trans`, que os gera pela própria função de transição estendida de `G`; dessarte, todos os estados marcados por `det_trans` são conjuntos compostos de estados originais do AFN `G`, sendo, ao menos um destes, final.

Apesar de o raciocínio ser ágil nesta prova, a demonstração assistida por computador não é tão simples e envolve diversas induções. O material dessas provas completas encontramos nos artefatos produzidos pelo presente trabalho no assistente Coq.

6.3.3 Alcançabilidade dos estados

A etapa que postergamos na prova de que todos os estados do AFD retornado pelo algoritmo de Brzozowski são alcançáveis é a que versa esta seção. Argu-

mentamos que, se a determinização só gera estados alcançáveis, então os estados resultantes do método de Brzozowski também são todos alcançáveis, notadamente.

Assim como fizemos na Subseção 6.3.2, precisamos destrinchar a concatenação feita pela função `det`. No caso em que não há estados finais no autômato original, o resultado é automaticamente vazio, uma constatação espontânea. Da concatenação, somente o lado esquerdo agora é útil para a demonstração. Mais uma vez conseguimos eliminar a normalização, já que ela mantém os caminhos originais, apenas possivelmente adicionando outros. Assim, se todos os estados `Q` de `G` são tais que

```
∃ w, path G' Q0 Q w
```

para um dado `Q0` – em nosso caso específico, `start_states G` –, então esses caminhos também existirão em `normalize G'` com a possível substituição, não obstante, de `Q0` e `Q` por equivalentes na normalização. No entanto, como `start Q0` é o primeiro termo de

```
G' = start (start_states G)::det_trans G (2^|states G|) (start_states G)
```

`Q0` não será substituído. O presente trabalho fez a demonstração dessa propriedade da normalização por indução sobre `w`.

A prova de que todos os estados de `G'` são alcançáveis fazemos pela indução do combustível `2^|states G|`. No passo, podemos eliminar `start (start_states G)` da conclusão, pois não interfere nos caminhos do AFN, apenas as transições afetam os caminhos. Por conseguinte, se

```
Q ∈ (states (det_trans G (S n) Q0))
```

então temos, três possibilidades: (1) `Q = Q0`, (2) `Q` é vizinho direto de `Q0` ou

```
Q ∈ (states (det_trans G n (δ̂ Q0 [a])))
```

para algum termo `a`. Os casos (1) e (2) são triviais, visto que o caminho está dado. Tendo a hipótese indutiva

```
∀ Q', ∃ w, path (det_trans G n Q') Q' Q w
```

Com `Q' = δ̂ Q0 [a]`, concluímos

```
path (det_trans G (S n) Q0) Q0 Q (a::w)
```

ou seja, `Q` é alcançável.

6.3.4 Determinismo

Obviamente, o algoritmo de determinização precisa garantir o que ele promete: o determinismo. Se não há estados finais no autômato de origem, o resultado é um AFD vazio. Caso contrário, destrinchemos outra vez a concatenação da normalização do estado inicial e as transições com a marcação dos estados finais. Sabemos que os estados finais não põem em cheque nenhuma das restrições do determinismo, assim pulemos tal marcação. Aqui, outrossim, necessitamos provar que `G' = start (start_states G)::det_trans G (2^|states G|) (start_states G)` e `normalize G'` são AFDs.

O autômato `G'` é determinístico porque só tem um estado inicial – `start_states G` , pois os demais são componentes construídos por `trans` – e todas as transições são determinadas pelo estado de origem e o símbolo; isto é, para toda transição `trans Q a Q' ∈ G'` , o estado `Q'` é determinado: `Q' = δ̂ G Q [a]` . Ao aplicarmos a normalização em `G'` , mantemos os construtores intactos; logo, o autômato resultante também só tem um estado inicial. Ademais, as transições `trans Q a Q' ∈ (normalize G')` também são determinadas:

```
Q' = normalize' (δ̂ G Q [a])
```

em que `normalize'` é a função que normaliza o estado de entrada, substituindo-o pelo primeiro conjunto equivalente do AFN `G'` . Portanto, `normalize G'` é, de fato, um AFD.

6.3.5 Relação entre caminhos

Nesta subseção, concentramos os esforços para mostrar que

```
(∀ Q', (q ∈ Q' -> path (det G) Q Q' w -> ∃ p, (p ∈ Q /\ path G p q w)))
/\
∀ p, (p ∈ Q -> path G p q w -> ∃ Q', (q ∈ Q' /\ path (det G) Q Q' w))
```

é verdadeira para quaisquer `q` e estado `Q`, como supomos que seria. Pelo fato de o lema ser uma conjunção, podemos dividir a prova em dois casos. No da esquerda, façamos indução sobre `path (det G) Q Q' w` para demonstrar o passo

```
∃ p, (p ∈ Q /\ path G p q (w ++ [a]))
```

Temos que `q ∈ Q''` e `trans Q' a Q'' ∈ (det G)`, então, dado que `Q'' = δ̂ G (normalize' Q') [a]` para algum `a`, deve existir um estado `t ∈ Q'` que seja vizinho de `q` por `a`. A hipótese indutiva

```
∀ t, (t ∈ Q' -> ∃ p, (p ∈ Q /\ path G p t w))
```

permite-nos concluir o passo indutivo.

Para o lado direito do lema, consideremos `Q = start_states G` e `|w| <= 2^|states G|`. Por indução sobre `2^|states G| = n` em `det_trans G (2^|states G|) (start_states G)`, conseguimos chegar à conclusão que desejamos. Na base, o caminho deve ser vazio; no passo, temos `w = a::w'` e a hipótese indutiva

```
∀ w' t Q0, (t ∈ Q0 -> |w'| <= n -> path G t q w' ->
    ∃ Q', (q ∈ Q' /\ path (det_trans G n Q0) Q0 Q' w'))
```

que nos propicia a dedução de

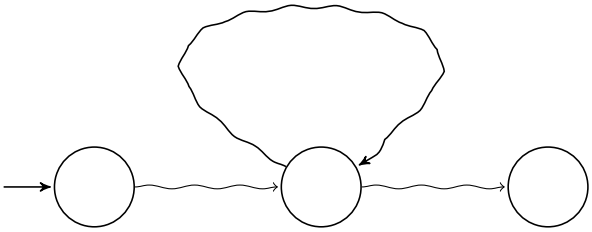
```
q ∈ Q' /\ path (det_trans G (S n) Q) Q Q' (a::w')
```

porque `|a::w'| = S |w'| <= S n`, `path G t q w'` com algum estado `t ∈ (δ̂ Q [a])` – já que `path G p q (a::w')`, `p ∈ Q` e `t ∈ (δ q a)` – e, consequentemente,

```
q ∈ Q' /\ path (det_trans G n (δ̂ Q [a]))
```

Com base na Subseção 6.3.3, todos os estados de `det G` são alcançáveis. Assim, podemos generalizar o lema para qualquer estado `Q` de `det G`, considerando agora `|wQ ++ w| <= n`, em que `path (det G) (start_states G) Q wQ`. Isso também nos permite concluir que todas as palavras de tamanho até `n` um aceita se e somente se o outro aceita.

Figura 9 – Lema do bombeamento



Fonte: Elaborada pelo autor, 2021.

Segundo o lema do bombeamento, todo caminho que percorre uma palavra de tamanho maior ou igual ao número de estados do AFN possui loops, conforme a Figura 9. Isso significa que podemos repartir o caminho em caminhos menores. O caminho

```
c1 ++ [q] ++ c2 ++ [q] ++ c3
```

existe se e somente se existem os caminhos

```
c1 ++ [q] ++ c3
```

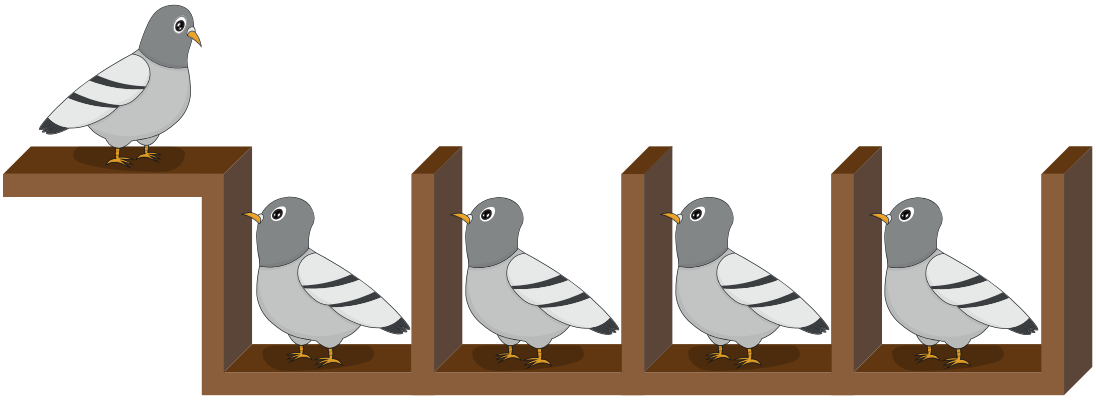
e

```
c1 ++ [q] ++ c2 ++ [q]
```

Fazendo essa repartição de diferentes modos para estes caminhos e, de forma indutiva, assim por diante, sobrariam caminhos de tamanho menor ou igual ao número de estados do autômato mais um, com no máximo um loop. Como toda palavra percorrida em um caminho tem o tamanho dele menos um, nosso AFD resultante `det G` tem até `2^|states G|` estados e todos os caminhos dessa ordem permitem deduzir o lema exposto, então qualquer caminho o faz devido ao lema do bombeamento.

O lema do bombeamento é uma conclusão direta do princípio da casa dos pombos (Figura 10), segundo o qual, tendo `n < m` casas para alocarmos `m` pombos, haverá necessariamente casas com mais de um pombo colocado (PIERCE et al., 2010). Desta analogia deduzimos que, para alocar os símbolos de uma transição estendida a um número `n <= |w|` de estados, haverá repetição de estados. Como ao primeiro estado alocamos o vazio, a comparação `<=` é devida.

Figura 10 – Princípio da casa dos pombos



Fonte: Elaborada pelo autor, 2021.

O presente trabalho, é importante ressalvar, não demonstrou a generalização dos caminhos exposta mediante a assistência computacional. Esta é uma tarefa que ficará para os trabalhos futuros.

6.4 CONSIDERAÇÕES DO CAPÍTULO

No capítulo atual, atestamos que o algoritmo de Brzozowski funciona de forma a obter um AFD sem estados equivalentes e com a linguagem revertida quando certas propriedades a reversão e determinização apresentadas respeitam. Argumentamos que a determinização mantém a linguagem intacta do autômato de entrada e a reversão reverte a linguagem. Também demonstramos que todos os estados do resultado da determinização são alcançáveis, que é uma exigência da minimização. Provamos que a determinização realmente produz um autômato determinístico. Por fim, apoiados nos artefatos produzidos com o Coq, tratamos da prova de alguns lemas relacionados à observação básica feita por Jan (2019). Isso nos permitiu concluir, no início deste capítulo, que o algoritmo de Brzozowski não gera estados indistinguíveis.

Diante do exposto, apontamos, com base em provas desenvolvidas no assistente de provas supracitado, o corretismo do algoritmo de Brzozowski para a minimização. Funções baseadas nas apresentadas podemos desenvolver em outras linguagens e paradigmas de programação, de forma a manter a lógica e otimizar as estruturas de dados, buscas e demais definições. A minimização proposta por Brzozowski e Tamm (2014) com

```
fun G => det (rev (det (rev G)))
```

definitivamente resulta em um AFD mínimo e equivalente ao AFD ao qual aplicamos esta função, com uma ressalva. Nossa definição de AFD não impõe a necessidade de haver um estado inicial, embora Brzozowski e Tamm (2014), Cassandras e Lafortune (2008), Hopcroft, Motwani e Ullman (2006) o façam. Por isso, o resultado `det (rev []) = []` é diferente de `det (rev [start q0]) = [start []]`, apesar de ambos terem a mesma linguagem vazia. Em nenhum dos casos existem estados inalcançáveis ou pares de estados equivalentes. Mesmo assim, `[]` é mínimo, e `[start []]` não. Com a imposição de estado inicial, o primeiro AFD resultante resta inválido; por isso, encontramos essa aparente contradição. Podemos resolver esse impasse adicionando uma condicional em nosso algoritmo de modo a retornar o vazio na inexistência de estados finais. Portanto, o método de Brzozowski cumpre o seu papel minimizador.

7 CONSIDERAÇÕES FINAIS

Este trabalho de conclusão de curso empregou definições não canônicas para a representação de AFNs em ambiente computacional, especificamente o assistente de provas Coq, que comprovamos ser eficiente e robusto para os objetivos postos. Depois, tendo inspiração nos trabalhos relacionados e com assistência desse sistema, provamos que o algoritmo de Brzozowski realmente cumpre o seu papel minimizador – retorna um AFD mínimo com linguagem reversa. Demonstramos que é possível provar diversos teoremas sobre AFNs sem a teoria dos conjuntos, empregando apenas a teoria dos tipos e estruturas de dados conhecidas, como listas. As principais contribuições do trabalho são uma demonstração do algoritmo de Brzozowski mediante conceitos matemáticos palpáveis e uma base de estudos para futuras provas assistidas por computador no que tange às propriedades de AFNs. O presente trabalho desenvolveu 3502 linhas de código na linguagem do Coq.

Para trabalhos futuros, podemos propor a demonstração de que, se e somente se um AFD não possui estados inalcançáveis e indistinguíveis, então não existe AFD equivalente com número menor de estados distintos: uma definição mais intuitiva de minimização. Seria interessante essa prova em Coq, uma vez que os trabalhos relacionados geralmente partem da mesma noção de AFD mínimo do presente trabalho. Outro ponto de destaque concerne à prova assistida de que o combustível do algoritmo de determinização deste trabalho é suficiente. Agregaria valor uma demonstração para tanto nos moldes e ferramentas dos artefatos produzidos.

Salientamos as diversas aplicações envolvendo AFDs, como a modelagem de sistemas a eventos discretos. Esses autômatos não são restritos aos formalismos da teoria da computação, mas servem de abstração e estrutura para aplicações práticas além das presentes no cotidiano do cientista da computação. Nesse sentido, trabalhos futuros acerca de tais utilidades são possíveis e de relevância.

REFERÊNCIAS

AARTS, F.; VAANDRAGER, F. Learning i/o automata. In: SPRINGER. **International Conference on Concurrency Theory**. 2010. p. 71–85. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-642-15375-4_6>. Acesso em: 16 jul. 2021.

ARISTÓTELES. **Organon**. Tradução de Pinharanda Gomes. Lisboa: Guimarães Editores, 1985. v. 1.

ATHALYE, A. **CoqIOA: a formalization of IO automata in the Coq proof assistant**. Tese (Doutorado) — Massachusetts Institute of Technology, 2017. Disponível em: <https://pdos.csail.mit.edu/papers/aathalye-meng.pdf>. Acesso em: 28 jun. 2021.

BARENDREGT, H.; GEUVERS, H. Proof-assistants using dependent type systems. In: **Handbook of automated reasoning**. University of Oregon, 2001. p. 1149–1238. Disponível em: <https://www.cs.uoregon.edu/research/summerschool/summer02/lectures/ts_read.pdf>. Acesso em: 7 ago. 2021.

BARRAS, B. et al. **The Coq proof assistant reference manual**. Rocquencourt: INRIA, 1999. v. 6. Disponível em: <http://flint.cs.yale.edu/cs430/coq/pdf/Reference-Manual.pdf>. Acesso em: 12 ago. 2021.

BLOT, A.; DAGAND, P.-É.; LAWALL, J. From sets to bits in Coq. In: **International Symposium on Functional and Logic Programming**. Springer, 2016. p. 12–28. Disponível em: <https://hal.inria.fr/hal-01251943v1>. Acesso em: 9 jul. 2021.

BOGDANOV, A. **Formal Languages and Automata Theory: Lecture 7**. The Chinese University of Hong Kong, 2010. Disponível em: <http://www.cse.cuhk.edu.hk/~andrejb/csc3130/f10/notes/10N07.pdf>. Acesso em: 4 ago. 2021.

BONCHI, F. et al. Brzozowskis algorithm (co) algebraically. In: **Logic and Program Semantics**. Springer, 2012. p. 12–23. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-642-29485-3_2>. Acesso em: 28 jun. 2021.

BONDY, J. A.; MURTY, U. S. R. et al. **Graph theory with applications**. London: Macmillan London, 1976. v. 290.

BRAIBANT, T.; POUS, D. Deciding Kleene algebras in Coq. **arXiv preprint arXiv:1105.4537**, 2011. Disponível em: <https://arxiv.org/abs/1105.4537>. Acesso em: 28 jun. 2021.

BRZOZOWSKI, J.; TAMM, H. Theory of átomata. In: **Theoretical Computer Science**. Elsevier, 2014. v. 539, p. 13–27. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0304397514002953>. Acesso em: 4 ago. 2021.

CASSANDRAS, C. G.; LAFORTUNE, S. **Introduction to discrete event systems**. 2. ed. New York: Springer Science+Business Media, 2008.

DEVLIN, K. **Fundamentals of contemporary set theory**. New York: Springer-Verlag, 1979.

DOCZKAL, C.; KAISER, J.-O.; SMOLKA, G. A constructive theory of regular languages in Coq. In: **International Conference on Certified Programs and Proofs**. Springer, 2013. p. 82–97. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-319-03545-1_6>. Acesso em: 28 jun. 2021.

FERREIRA, A. B. de H. **Minidicionário Aurélio**. 8. ed. São Paulo: Positivo, 2018.

GONTHIER, G. et al. Formal proof–the four-color theorem. In: **Notices of the AMS**. American Mathematical Society, 2008. v. 55, n. 11, p. 1382–1393. Disponível em: <https://www.sanitas.es/media/sanp/documento/ver-las-tarifas-2019/georges-gonthier.pdf>. Acesso em: 12 ago. 2021.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. 3. ed. Boston: Pearson/Addison Wesley, 2006.

JAN, H. **Proof of Brzozowski's algorithm for DFA minimization**. Computer Science Stack Exchange, 2019. Disponível em: <https://cs.stackexchange.com/questions/105574/proof-of-brzozowskis-algorithm-for-dfa-minimization>. Acesso em: 17 maio 2021.

LAFORTUNE, S. Discrete event systems: Modeling, observation, and control. **Annual Review of Control, Robotics, and Autonomous Systems**, Annual Reviews, v. 2, p. 141–159, 2019. Disponível em: <https://www.annualreviews.org/doi/abs/10.1146/annurev-control-053018-023659>. Acesso em: 18 out. 2019.

LUO, Z. **Computation and reasoning: a type theory for computer science**. Oxford New York: Clarendon Press Oxford University Press, 1994.

MARASS, F.; UPTON, C. Sequence searcher: A java tool to perform regular expression and fuzzy searches of multiple dna and protein sequences. **BMC research notes**, BioMed Central, v. 2, n. 1, p. 1–3, 2009. Disponível em: <https://bmcresnotes.biomedcentral.com/articles/10.1186/1756-0500-2-14>. Acesso em: 15 jul. 2021.

MENEZES, P. B. **Linguagens formais e autômatos**. 3. ed. Porto Alegre: Sagra-Dcluzzato, 2000.

MOURA, L. de et al. The lean theorem prover (system description). In: **International Conference on Automated Deduction**. Springer, 2015. p. 378–388. Disponível em: <https://www.springerprofessional.de/en/the-lean-theorem-prover-system-description/2500952>. Acesso em: 7 ago. 2021.

PAULSON, L. C. A formalisation of finite automata using hereditarily finite sets. In: SPRINGER. **International Conference on Automated Deduction**. 2015. p. 231–245. Disponível em: <https://arxiv.org/pdf/1505.01662.pdf>. Acesso em: 28 jun. 2021.

PIERCE, B. C. et al. **Software foundations**. Philadelphia: Penn CIS, 2010. Disponível em: <https://softwarefoundations.cis.upenn.edu>. Acesso em: 31 mar. 2020.

RICHMAN, F. Interview with a constructive mathematician. **Modern Logic**, The Review of Modern Logic, v. 6, n. 3, p. 247–271, 1996. Disponível em: <https://projecteuclid.org/journals/modern-logic/volume-6/issue-3/Interview-with-a-constructive-mathematician/rml/1204835729.pdf>. Acesso em: 8 ago. 2021.

SAVCHENKO, S. Practical regular expression mining and its information quality applications. In: **ICIQ**. M.I.T. Information Quality Program, 2002. p. 177–186. Disponível em: <http://mitiq.mit.edu/ICIQ/Documents/IQ%20Conference%202002/Papers/PracticalRegularExpressionMiningnItsInfoQualityApp.pdf>. Acesso em: 15 jul. 2021.

SETZER, A. Java as a functional programming language. In: SPRINGER. **International Workshop on Types for Proofs and Programs**. 2002. p. 279–298. Disponível em: <https://link.springer.com/chapter/10.1007/3-540-39185-1_16>. Acesso em: 7 ago. 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.

STROUSTRUP, B. What is object-oriented programming? **IEEE software**, IEEE, v. 5, n. 3, p. 10–20, 1988. Disponível em: <https://ieeexplore.ieee.org/abstract/document/2020>. Acesso em: 12 ago. 2021.

TABAKOV, D.; VARDI, M. Y. Experimental evaluation of classical automata constructions. In: **International conference on logic for programming artificial intelligence and reasoning**. Springer, 2005. p. 396–411. Disponível em: <https://www.cs.rice.edu/~vardi/papers/lpar051.pdf>. Acesso em: 12 ago. 2021.

VASILIADIS, G. et al. Regular expression matching on graphics hardware for intrusion detection. In: SPRINGER. **International Workshop on Recent Advances in Intrusion Detection**. 2009. p. 265–283. Disponível em: <https://link.springer.com/chapter/10.1007/978-3-642-04342-0_14>. Acesso em: 15 jul. 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.