Marcos Vinicius Moreira Serra Benevides

Implementação e análise de uma engine para expressões regulares em Coq via testes baseados em propriedades

São Luís - MA, Brasil

22 de julho de 2019

Marcos Vinicius Moreira Serra Benevides

Implementação e análise de uma engine para expressões regulares em Coq via testes baseados em propriedades

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, **como parte dos requisitos necessários** para obtenção do grau de Bacharel e Ciência da Computação.

Universidade Federal do Maranhão – UFMA

Departamento de Informática – DEINF

Ciência da Computação

Orientador: Sérgio Souza Costa

Coorientador: Rodrigo Geraldo Ribeiro

São Luís - MA, Brasil

22 de julho de 2019

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Núcleo Integrado de Bibliotecas/UFMA

Vinicius Moreira Serra Benevides, Marcos.
Implementação e análise de uma engine para expressões regulares em Coq via testes baseados em propriedades / Marcos Vinicius Moreira Serra Benevides. - 2019.
67 p.

Coorientador(a): Rodrigo Geraldo Ribeiro.
Orientador(a): Sérgio Souza Costa.
Monografia (Graduação) - Curso de Ciência da Computação, Universidade Federal do Maranhão, São Luís, 2019.

1. Coq. 2. Expressões Regulares. 3. Programação Funcional. I. Geraldo Ribeiro, Rodrigo. II. Souza Costa, Sérgio. III. Título.

Marcos Vinicius Moreira Serra Benevides

Implementação e análise de uma engine para expressões regulares em Coq via testes baseados em propriedades

Monografia apresentada ao curso de Ciência da Computação da Universidade Federal do Maranhão, **como parte dos requisitos necessários** para obtenção do grau de Bacharel e Ciência da Computação.

Trabalho aprovado. Nota: ___,___

Sérgio Souza Costa
Orientador

Rodrigo Geraldo Ribeiro
Coorientador

Davi Viana dos Santos

Luciano Reis Coutinho

São Luís - MA, Brasil
22 de julho de 2019

Aos amigos que fiz no Núcleo de Computação Aplicada
e no Laboratório de Mídias Interativas.
Aos professores Ivaldo Paz Nunes e Márcios Kléos,
que me acolheram, respectivamente,
no Departamento de Matemática e no
Grupo de Estudos e Lógica e Filosofia Formal.
E, por último, aos meus familiares.
Minha avó Jacira, meu pai José,
minhas mães Carmen e Naura,
Minhas irmãs Camila e Vitória,
Meu irmão Daniel e
meu falecido irmão, Lucas.

Agradecimentos

Aos meus orientadores Sérgio Souza Costa e Rodrigo Geraldo Ribeiro, que disponibilizaram tempo, suporte, recursos e especialmente paciência com os meus atrasos e erros, agradeço a vocês por tudo.

"Consider ye the seed from which ye sprang;
Ye were not made to live like unto brutes,
But for pursuit of virtue and of knowledge."
(Dante Alighieri, Inferno - Canto XXVI)

Resumo

Um provador interativo de teoremas (ou assistentes de provas) é uma ferramenta que auxilia o desenvolvimento de provas formais. Este trabalho fez uma revisão bibliográfica dos fundamentos matemáticos destas ferramentas. Além disso apresentou o assistente de provas COQ e como ele implementa alguns destes conceitos. Alternativamente, apresentou-se os testes baseados em propriedades que é uma solução intermediária entre os testes unitários e as provas formais. Este trabalho utilizou então essa metodologia para analisar um algoritmo de expressões regulares que foi publicado como *Functional pearl*.

Palavras-chave: coq. expressões regulares. programção funcional. provadores de teoremas. quickcheck. testes baseados em propriedades.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

Match	De "Matching", correspondência.
Regex	Uma abreviação do termo "Regular Expression" (Expressão Regular).
ADT	Abreviação para o termo "Algebraic Data Types" (Tipos de Dados Algébricos).

Lista de símbolos

α	Letra grega Alfa
β	Letra grega Beta
Γ	Letra grega Gama
Δ	Letra grega Delta
η	Letra grega Eta
λ	Letra grega Lambda
$\in$	Pertence
$\cup$	União de conjuntos
$\bot$	Constante lógica (*Falsum*) denotando "Falso"
$\top$	*Verum*, oposto de $\bot$
$\neg$	Símbolo lógico para a *negação*
$\vdash$	Consequência lógica

Sumário

1 Introdução

> *'**Perlis:** Boa parte da complexidade de um programa é espúria, vários casos testes podem ser estudados para exaurir o problema. A dificuldade trata-se apenas de isolar os melhores casos de teste, não em provar o algoritmo, pois isso segue da escolha dos melhores casos de testes.*
> ***Dijkstra:** Testes apenas demonstram a presença, não a ausência, de bugs.'*
> (RANDELL, 1996)

A área de computação denominada de métodos formais utiliza-se de conceitos fundamentados na matemática e lógica na especificação de construção de software e hardware. Esta área de conhecimento é extremamente importante para a computação distribuída, dado a complexidade de seus protocolos que requer especificações detalhadas.

A robustez e a qualidade de um software pode estar atrelada diretamente a quão bem escritos estão seus testes. Uma base de código não testada torna difícil a manutenção e a evolução do software, dado que as alterações podem provocar erros que serão difíceis de encontrar e isolar. Então, testar código não é só importante, mas é algo necessário ao desenvolvimento de qualquer sistema. Contudo, como qualquer código, testes exigem manutenção e evolução. Testes ruins e ineficientes podem tornar uma integração contínua inviável. Portanto, é desejável que se escrevam testes mais robustos (que testam mais) com a menor quantidade de código possível.

De forma geral, um projeto de software genérico possuirá três níveis: testes unitários, testes de integração e testes de sistema (DOOLEY, 2011). Testes unitários por sí só não garantem a corretude de um sistema como um todo, apenas de um determinado módulo. Deve-se também garantir que os testes do módulo foram bem pensandos, i.e., eles testam situações inusitadas ou casos pouco comuns. A responsabilidade de escrever bons testes fica a cargo da equipe de desenvolvimentos. Os testes de integração buscam garantir que a comunicação entre os módulos está sendo feita de forma correta, i.e., os testes são escritos com informações providas pelas interfaces, sendo desnecessário conhecer como os módulos estão implementados. O testes de sistema verificam se o sistema está completamente integrado de acordo com o que é esperado.

Como já foi enfatizado na citação de Dijkstra, testes não garantem a ausência de bugs, apenas uma prova formal pode fazê-lo. Contudo, provas matemáticas da corretude de software podem levar tempo e exigem um tipo de conhecimento muito específico. Nesse ponto, a utilização de testes baseados em propriedades podem ser uma boa alternativa, pois não só oferecem garantias melhores que as providas por testes unitários.

A plataforma utilizada para este trabalho é o assistente de provas Coq (TEAM, 2019), que foi criado para desenvolver provas matemáticas, escrever especificações formais de programas e verificar sua corretude com respeito à especificação. A linguagem usada para escrever as provas chama-se Gallina[1]. Os termos dessa linguagem podem representar programas, propriedades e provas de propriedades. Usando o Isomorfismo de Curry-Howard (seção 2.1.3) programas, propriedades e provas podem ser formalizadas na mesma linguagem, conhecida como "Cálculo de Construções Indutivas", trata-se de um λ-Cálculo tipado de ordem superior, inicialmente investigado em (COQUAND; HUET, 1986). A lógica central é de natureza intuicionista (seção 2.1.1), regras como terceiro excluído ($\vdash P \vee \neg P$) ou eliminação da dupla negação ($\neg\neg P \vdash P$) não valem naturalmente. Alguns projetos de notoriedade feitos em COQ incluem:

- CompCert (LEROY et al., 2012): Um compilador para um subconjunto da linguagem C formalmente verificado, o código gerado em assembly tem garantias de seguir o comportamento descrito pelo código fonte em C.
- Uma prova formal do famoso teorema das 4 cores (GONTHIER, 2008).
- CertiKOS (GU et al., 2011): Uma arquitetura para construção de kernels concorrentes.
- FSCQ (CHEN et al., 2017): Um sistema de arquivos formalmente verificado.
- O projeto UniMath (VOEVODSKY et al., 2014): Uma biblioteca que visa formalizar um corpo substancial da matemática sob o ponto de vista univalente.

Recentemente, o ecosistema do COQ ganhou uma implementação do QuickCheck (CLAESSEN; HUGHES, 2011) denominada QuickChick. O QuickCheck é um protocolo originalmente implementado em Haskell e também trata-se da primeira biblioteca para testes baseados em propriedaes. A utilização de uma biblioteca para testes num provador de teoremas pode parecer estranha, mas é extremamente útil quando lidados com conjecturas, se um caso que as falsifique for encontrado então podemos usar essa informação para melhorar nossas definições ou tentar entender melhor o problema, evitando assim a perda de tempo numa prova impossível.

1.1 Objetivo geral

- Implementar e analisar uma engine para expressões regulares através de testes baseados em propriedades.

[1] Um trocadilho com Coq, que significa "Galo" em francês

1.2 Objetivos específicos

- Fazer uma revisão bibliográfica dos fundamentos matemáticos e lógicos para criação de software correto por construção.
- Implementar o algoritmo descrito no artigo "A play on regular expressions" em Coq.
- Implementar geradores e verificar a robustez da implementação em COQ usando testes baseados em propriedades da biblioteca QuickChick.

2 Fundamentação Teórica

'Aristóteles formulou seus silogismos na antiguidade e William de Ockham estudou lógica na idade média. Contudo, a disciplina de lógica moderna começou com Gottlob Frege e seu Begriffschrift, escrito em 1879 quando o mesmo tinha 31 anos.' Philip Wadler (WADLER, 2000)

2.1 A lógica do séc.XIX e a crise do séc.XX

Friedrich Ludwig Gottlob Frege (1848 - 1925) foi um matemático, lógico e filósofo alemão que trabalhou na Universidade de Jena. Fregue essencialmente reestruturou a disciplina da Lógica ao construir um sistema formal que deu origem ao cálculo de predicados. Em seu sistema formal, Frege desenvolveu a análise de proposições quantificadas e formalizou a noção de "Prova" em termos que ainda são aceitos atualmente. Frege então demonstrou que seria possível utilizar este sistema para resolver proposições matemáticas teóricas em termos de noções lógicas e matemáticas mais simples. Um dos Axiomas que Frege adicionou ao seu sistema, na tentativa de derivar partes significantes da Matemática via Lógica, acabou sendo incosistente (ZALTA, 2018).

Figura 1 – Gottlob Frege e Bertrand Russell

O paradoxo foi uma descoberta independente do filósofo inglês Bertrand Russell e acabou demonstrando que o formalismo desenvolvido por Frege era, na verdade, inconsistente. Particurlamente o problema se encontrava no Axioma V, conhecido atualmente como compreensão irrestrita de conjuntos (IRVINE; DEUTSCH, 2016). O paradoxo de Russell se origina da ideia de que qualquer condição pode ser usada para determinar um conjunto:

1. A notação da compreensão de conjuntos, denotando "O conjunto dos x tal que $\varphi(x)$".

$$\{x \mid \varphi(x)\}$$

2. Seja R o conjunto $\{x \mid \neg\varphi(x)\}$, onde $\varphi(x) = x \in x$.

3. $R \in R$?

4. Paradoxo:

   - Se $R \in R$, então $R \notin R$.
   - Se $R \notin R$, então $R \in R$.

   ou seja:

$$R \in R \iff R \notin R$$

2.1.1 O programa de Hilbert, Gödel e Brouwer

As respostas a este paradoxo começaram a surgir no início do século XX. O *Principia Mathematica* de Russell & Whitehead demonstrou que o formalismo lógico poderia, de fato, expressar boa parte da Matemática. Inspirado por essa visão, David Hilbert e seus colegas em Göttigen propuseram um programa para a construção de um base axiomática para a Lógica e a Teoria dos Conjuntos, de forma a resolver o *Entscheidungsproblem*[1], i.e., desenvolver um procedimento *efetivamente calculável* para determinar a verdade ou falsidade de qualquer proposição. Isso, porém, pressupõe a noção de completude: Para cada afirmação P, ou P ou sua negação ($\neg P$) possuem uma prova. O programa liderado por Hilbert foi enfraquecido pelos resultados obtidos por Kurt Gödel em 1930, onde o mesmo provou que a aritmética era incompleta (WADLER, 2015).

Paralelamente o matemático holandês Luitzen Brouwer desenvolveu o intuicionismo, cuja idéia básica é a de que não se pode afirmar a existência de um objeto matemático a menos que se possa definir um procedimento para construí-lo. Os matemáticos Arend Heyting e Andrey Kolmogorov formalizaram essa ideia de forma idependente nos anos 30. A interpretação BHK[2] atribui um tipo diferente de interpretação para o ato de provar. Na lógica clássica a semântica de uma fórmula pode ser analizada através de sua tabela verdade, na lógica intuicionista utiliza-se a ideia de *construção* de uma prova, que pode ser especificada via indução na estrutura das fórmulas (TROELSTRA, 2003):

- Uma prova de $A \wedge B$ é dada por um par de provas $\langle p, q\rangle$, onde p é uma prova de A e q é uma prova de B.

[1] Problema da decisão
[2] Brower, Heyting & Kolmogorov

- Uma prova de $A \vee B$ possui forma $\langle 0, p\rangle$ (sendo p uma prova de A) ou $\langle 1, q\rangle$ (onde q é uma prova de B).
- A prova de $A \to B$ é uma construção q que transformar qualquer prova p de A em uma prova $q(p)$ de B.
- $\bot$ não possui prova. A prova de $\neg A$ é uma construção que transforma qualquer prova de A numa prova de $\bot$, i.e., $\neg A$ é definida como $A \to \bot$.
- Uma prova p de $\exists x \in D, \varphi(x)$ é um par $\langle d, q\rangle$, onde $d \in D$ e q é uma prova de $\varphi(d)$.
- Uma prova p de $\forall x \in D, \varphi(x)$ é uma função que converte qualquer elemento $d \in D$ numa prova de $\varphi(d)$.

2.1.2 Church, Turing e Gentzen

Alonzo Church introduziu o λ-Cálculo como um novo formalismo para os fundamentos da matemática, muitos o consideram a primeira linguagem de programação, criada décadas antes dos primeiros computadores. Essa formulação também é capaz de representar qualquer função computável. De forma independente (e no mesmo ano) Alan Turing escreveu um artigo sobre a máquina que recebe seu nome, demonstrando que o *Entscheidungsproblem* não é decidível e que sua formulação era equivalente à de Church (WADLER, 2000).

Um tratamento mais rigoroso sobre o cálculo de Church pode ser encontrando em (HINDLEY; SELDIN, 2008), nesta seção procura-se desenvolver algumas noções forma intuitiva. No λ-Cálculo a noção de computação é expressa através da abstração e aplicação de funções. Dado um conjunto contável de símbolos X, um λ-termo pode ser definido da seguinte forma indutiva como uma:

- Variável: Se $e \in X$, então e é um λ-termo.
- Aplicação: Sejam e_1 e e_2 λ-termos, então a aplicação $(e_1\, e_2)$ é um λ-termo.
- Abstração: Dada uma variável $x \in X$ e um λ-termo e, temos que $(\lambda x.e)$ também é um λ-termo.

No λ-Cálculo temos um universo onde *programas são dados*. A definição de aplicação trabalha com a ideia de que qualquer termo pode ser um programa ou um dado. Podemos aplicar qualquer termo f a qualquer termo t, gerando um novo termo $(f\, t)$ (que pode ser lido como uma função $f(t)$). Nas abstrações entende-se por $\lambda x.t$ um programa que, dado x como argumento, retorna t como saída, por exemplo, o programa $(\lambda x.(x\, x))$ toma x como entrada e retorna $(x\, x)$ como saída (BAEZ; STAY, 2010).

Dizemos que x é uma variável *ligada* se x está no escopo de algum λ, por exemplo x é ligado em $(\lambda x.yx)$, a variável y não está no escopo de nenhum λ em $(\lambda x.yx)$ e é dita ser *livre*. O conjunto de todas as variáveis *livres* de um λ-termo P é denotado por $FV(P)$ (HINDLEY; SELDIN, 2008). A semântica dos λ-termos é definida a partir de regras de redução/conversão:

- α-conversão: Permite a renomeação de variáveis que estejam no escopo de um λ.

$$(\lambda x.(xx)) \equiv_\alpha (\lambda y.(yy))$$

- β-redução: A aplicação de uma abstração $(\lambda x.t)$ no λ-termo s deve, necessariamente, retornar um t com todas as ocorrências de x substituidas por s $(t[s/x])$.

$$((\lambda x.t)\, s) \rhd_\beta t[s/x]$$

Exemplo 2.1.1.

$$(\lambda x.x(xy))\, n \rhd_\beta (x(xy))[x/n] = n(ny)$$

- η-redução: Elimina abstrações redundantes.

$$(\lambda x.(t\, x)) \rhd_\eta t,\ x \notin FV(t)$$

Exemplo 2.1.2.

$$(\lambda x.ex)\, e' \rhd_\eta e\, e'$$

O cálculo de Church apresenta inconsistências, inicialmente encontradas por Kleene e Rosser em (KLEENE; ROSSER, 1935), sendo depois simplificadas por Haskell Curry (SHAPIRO; BEALL, 2018). Isso levou Church a desenvolver versões tipadas deste sistema formal nos anos 40.

O segundo objetivo do programa de Hilbert foi o estabelecimento da consistência de várias Lógicas, uma Lógica inconsistente pode derivar qualquer fórmula, o que a torna inútil. Em 1935 o alemão Gerhard Gentzen introduziu não apenas uma, mas duas novas formulações para a lógica: Dedução Natural e o Cálculo com Sequentes. Gentzen generalizou a noção de *julgamento*, originalmente desenvolvida por Frege, para incluir a noção de Hipóteses em seus Sequentes, a notação:

$$\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_n$$

significa que a partir das premissas $\phi_1, \ldots, \phi_n$ pelo menos uma das fórmulas $\psi_1, \ldots, \psi_n$ é verdadeira. Se Γ e Σ são conjuntos de fórmulas, o sequente $\Gamma \vdash \Sigma$ indica que é consequência de todas as fórmulas de Γ que pelo menos uma fórmula em Σ é verdadeira. Sequentes básicos como $\phi \vdash \phi$ são tomados como axiomas. Em contraste com os sistemas dedutivos

de Hilbert, caracterizados por poucas regras de inferência e muitos axiomas, os sistemas de Gentzen possuem apenas um axioma e várias regras de inferências (com os conectivos lógicos).

A originalidade desses sistemas dedutivos está em perceber que regras lógicas devem vir sempre em pares (introdução e eliminação). Uma regra de introdução especifica sob quais circunstâncias é possível asserir uma fórmula dado um conectivo lógico, enquanto as regras de eliminação especificam como usar os conectivos lógicos para gerar conclusões (WADLER, 2015).

$$\frac{}{\Gamma, A \vdash A}\,\textsc{Id}$$

$$\frac{\Gamma, B \vdash A}{\Gamma \vdash B \to A} \to I \qquad \frac{\Gamma \vdash B \to A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A} \to E$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \wedge I \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge E_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge E_2$$

Figura 2 – Dedução Natural de Gentzen.

2.1.3 O isomorfismo de Curry-Howard

O isomorfismo (ou correspondência) de Curry-Howard extende a interpretação BHK numa relação entre programas e provas. Trata-se de observação que provas na Dedução Natural Intuicionista de Gentzen possuem uma correspondência com com programas escritos no λ-Cálculo tipado de Church (WADLER, 2015). Essa relação foi descoberta por Haskell B. Curry e William Howard.

$$\frac{}{\Gamma, x : A \vdash x : A}\,\textsc{Id}$$

$$\frac{\Gamma, x : B \vdash t : A}{\Gamma \vdash \lambda x.t : B \to A} \to I \qquad \frac{\Gamma \vdash t : B \to A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash t\, u : A} \to E$$

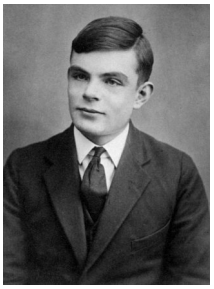
$$\frac{\Gamma \vdash t : A \quad \Delta \vdash u : B}{\Gamma, \Delta \vdash \langle t, u \rangle : A \wedge B} \wedge I \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t.\texttt{fst} : A} \wedge E_1 \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash t.\texttt{snd} : B} \wedge E_2$$

Figura 3 – λ-Cálculo tipado de Church (vermelho), Dedução Natural de Gentzen (azul).

A ideia de usar computadores para conduzir uma massiva formalização da matemática é bem antiga, mas só foi aparecer como um projeto bem articulado a partir dos esforços do matemático N.G. de Brujin, em seu Automath de Brujin utilizou o isomorfismo de Curry-Howard para verificar computacionalmente teoremas matemáticos. Nas palavras do próprio de Brujin (BRUIJN, 1983):

(a) Kurt Gödel

(b) Gerhard Gentzen



(c) Alan Turing

(d) L. E. J. Brouwer

(e) Arend Heyting

(f) Andrey Kolmogorov

(g) Alonzo Church

(h) Haskell B. Curry

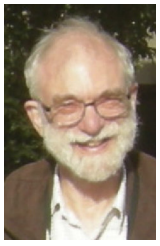
(i) William A. Howard

(j) Nicolaas de Bruijn

(k) Jean-Yves Girard

(l) John C. Reynolds

(m) Robin Milner



(n) J. Roger Hindley

Figura 4 – Durante o séc.XX, lógicos, matemáticos e cientistas da computação chegaram a resultados similares, utilizando abordagens diferentes.

> O *Automath* é uma linguagem para expressar pensamentos matemáticos de forma detalhada. Não se trata de uma linguagem de programação, embora de se assemelhar com uma. É definido através de uma gramática, todo texto escrito de acordo com suas regras têm garantia de se corresponder a um conceito matemático correto. Ele pode ser usado para expressar boa parte da matemática e admite diferentes fundamentos. As regras são tais que um computador por ser instruído para verificar se os textos escritos na linguagem estão corretos. Textos estes que não estão apenas restritos a provas de teoremas simples; podendo conter teorias matemáticas inteiros, incluindo as regras de inferência usadas em tais teorias.

Um exemplo real da correspondência de Curry-Howard é o aparecimento de "nomes-com-hífen" na literatura da Teoria dos Tipos.Vez ou outra um lógico motivado por problemas da Lógica e um cientista da computação motivado por problemas práticos acabam descobrindo exatamente o mesmo sistema de tipos (geralmente com o lógico chegando primeiro). A maioria das linguagens funcionais utiliza o sistema de tipos de Hindley-Milner, descoberto pelo lógico Hindley em 1969 e re-rescoberto pelo cientista da computação Milner em 1978. O mesmo ocorreu com os sistemas de Girard-Reynolds, descobertos pelo lógica Jean-Yves Girard em 1972 e re-rescobertos pelo cientista da computação John Reynolds em 1974 (WADLER, 2000).

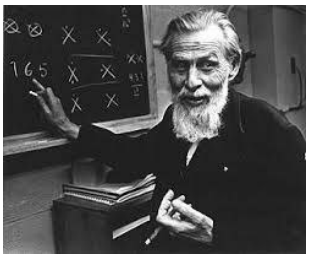
Juntos, todos esses projetos ajudaram a concentrar a atenção nas conexões entre lógica, linguagem e matemática. Eles também permitiram que os lógicos desenvolvessem uma consciência explícita da natureza dos sistemas formais e dos tipos de resultados metalógicos e metamatemáticos que provaram ser centrais para a pesquisa nos fundamentos da lógica e da matemática nos últimos cem anos (IRVINE; DEUTSCH, 2016).

A Ciência da Computação como disciplina nasceu dos formalismos desenvolvidos durante a crise dos fundamentos da matemática no século passado. A presença de conceitos lógicos ocupa uma parte central da disciplina, chegando a ser descrita como "o cálculo da ciência da computação" (MANNA; WALDINGER, 1985).

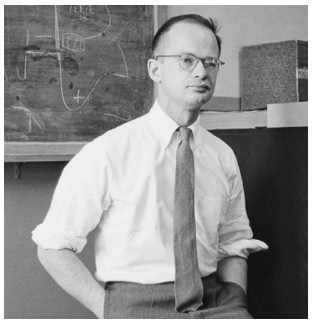
2.2 Expressões Regulares

Warren McCulloch e Walter Pitts em (MCCULLOCH; PITTS, 1943) desenvolveram modelos para tentar entender como cérebro humano conseguia produzir padrões complexos a partir de células básicas conectadas. Neste trabalho é descrito um modelo bem simplificado de um neurônio, sendo um dos primeiros exemplos do que veio a ser chamado depois de `redes neurais`.

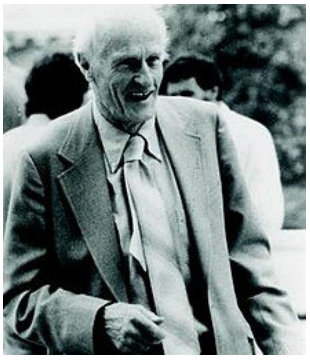
Stephen Kleene, um matemático americano e estudante de Alonzo Church, inspirado pelo trabalho de Warren e Pits descreveu estes modelos em (KLEENE, 1951) com uma álgebra que chamou de "conjuntos regulares". A notação usada para expressar esses conjuntos ficou conhecida como "expressões regulares", os resultados de Kleene, porém, só seriam aplicados na computação com o surgimento do Unix.



(a) Warren McCulloch



(b) Walter Pitts



(c) Stephen Coole Kleene

(d) Ken Thompson

Figura 5 – Entre as primeiras conjecturas de Warren e Pitts, até a primeira implementação por Thompson, se passaram mais de 20 anos

O Unix (oficialmente UNIX) é um sistema operacional multi-tarefa, multi-usuário que possui muitas variações ao longo do tempo. O Unix original foi desenvolvido nos laboratórios Bell da AT&T por Ken Thompson, Dennis Ritchie, entre outros. Thompson publicou em 1968 "Um Algoritmo para buscas via Expressões Regulares" (THOMPSON, 1968), depois implementando a notação de Kleene no editor de texto padrão do Unix (`ed`). O trabalho de Thompson inspirou a criação de compiladores para expressões regulares e eventualmente levou à criação to programa `grep` em 1973 e sua implementação mais completa (o `egrep`) em 1979, por Alfred Aho. Outras ferramentas do Unix, como o editor de streams `sed` e a linguagem `awk`, também provêm mecanismos para descrição de padrões via expressões regulares.

Símbolo	Função
`.`	Match em qualquer caractere
`[chars]`	Match em qualquer caractere no escopo dos colchetes
`[^chars]`	Match em qualquer caractere que não está no escopo dos colchetes
`^`	Match no início da linha
`$`	Match no fim de linha
`\w`	Match em qualquer "palavra", i.e., `[a-zA-Z0-9_]`
`\s`	Match em caracteres que representam espaços em branco, i.e., `[ \f\t\r\n]`
|	Match no elemento da esquerda ou da direita
`(expr)`	Agrupa elementos
?	Zero ou uma instância do elemento que a precede
$*$	Zero ou mais instâncias do elemento que a precede
$+$	Um ou mais instâncias do elemento que a precede
$\{n\}$	Match em exatas n instâncias de elementos que a precede
$\{\text{min},\}$	Match em no mínimo n instâncias
$\{\text{min}, \text{max}\}$	Match em qualquer quantidade de instâncias entre min e max

Tabela 1 – Tabela com metacarateres usados em *Perl Compatible Regular Expressions* nos Unixes (NEMETH, 2017).

```
$ echo "the quick brown fox" | sed 's/^/start /'
start the quick brown fox

$ echo "<b>Removing</b> those <b>HTML</b> tags." | sed 's/<[^>]*>//g'
Removing those HTML tags.
```

Listing 1 – Exemplos da utilização de regexes e do editor de streams `sed`.

2.3 Alfabetos, Linguagens e Strings

Um alfabeto é um conjunto finito (e não vazio) de caracteres, usualmente denotado por Σ. Os elementos de Σ são chamados de "símbolos". Alguns exemplos comuns de alfabetos são:

- $\Sigma_L = \{a, b, c, \ldots, z, A, B, \ldots, Z\}$ é o alfabeto latino padrão.
- $\Sigma_2 = \{0, 1\}$ é um alfabeto binário.

- O conjunto de todos os caracteres `ASCII`.

Uma `string` (ou `palavra`) é qualquer sequência finita de símbolos extraídos de um alfabeto específico. Por exemplo, "010101111" é uma string gerada pelos símbolos do alfabeto $\Sigma = \{0, 1\}$, a string "1111" também pode ser gerada pelo mesmo alfabeto.

2.3.1 Operações em strings

Uma operação útil quando se lida com strings é poder computar seu comprimento. O *comprimento* é definido pela quantidade de posições pros símbolos de uma string, vale ressaltar que isso não deve ser confundido com a quantidade de símbolos. Por exemplo, a string "1010" possui apenas 2 símbolos (0 e 1), mas seu comprimento, i.e. a quantidade de posições para estes símbolos, é 4. A notação padrão para o comprimento de uma string w qualquer é $|w|$.

Outra operação usual é poder tomar duas strings arbitrárias x, y pertencentes a um alfabeto Σ e poder concatená-las. A *concatenação* de duas strings x, y formadas de símbolos de Σ, denotada por xy, representa a composição dos símbolos de $x = a_1a_2 \dots a_{k-1}a_k$ com os símbolos de $y = b_1b_2 \dots b_{l-1}b_l$, formando $xy = a_1a_2 \dots a_{k-1}a_kb_1b_2 \dots b_{l-1}b_l$ de comprimento $k + l$.

Com as definições anteriores é possível trabalhar com a noção de uma "string vazia", usualmente denotada por ε, ε é resultado da escolha de 0 símbolos de qualquer alfabeto Σ. A string vazia possui propriedades interessantes como:

- ε tem comprimento 0.

$$|\varepsilon| = 0$$

- ε é o elemento neutro da concatenação.

$$\varepsilon w = w = w\varepsilon$$

Se Σ é um alfabeto, pode-se expressar o conjunto de todas as strings com um determinado comprimento usando a notação de exponenciação. Define-se Σ^k como o conjunto das strings geradas por Σ que possuem comprimento k. Por exemplo, dado $\Sigma = \{0, 1\}$, tem-se:

- $\Sigma^0 = \{\varepsilon\}$
- $\Sigma^1 = \{0, 1\}$
- $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^3 = \{000, 001, 010, 100, 101, 011, 110, 111\}$

O conjunto de todas as strings em um alfabeto Σ é denotado por Σ^*, se a string vazia for desconsiderada usa-se Σ^+. Ambos os casos são representandos por uma união infinita:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots \tag{2.1}$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots \tag{2.2}$$

2.3.2 Linguagens Regulares

Uma linguagem $\mathcal{L}$ sob um alfabeto Σ é um conjunto $\mathcal{L} \subseteq \Sigma$. Alguns exemplos abstratos incluem:

- Σ^* é uma linguagem sob qualquer alfabeto Σ.
- O conjunto $\{\varepsilon, 01, 10, 0011, 0011, 0101, 1010, \ldots\}$ é uma linguagem sob $\Sigma = \{0, 1\}$ que contém todas as strings de Σ^* com uma quantidade igual de zeros e uns.
- O conjunto $\{\varepsilon\}$ consiste apenas da string vazia e é uma linguagem sob qualquer alfabeto Σ.
- A linguagem $\mathcal{D}$ sob of alfabeto $\Sigma = \{0, 1\}$ contém todas as sequências infinitas formadas por 0 e 1.

$$\mathcal{D} = \{\omega_0\omega_1\omega_2 \ldots \mid \omega_i \in \{0, 1\}, \forall i \in \mathbb{N}\}$$

- $\emptyset \subset \Sigma^*$ é uma linguagem sob qualquer alfabeto Σ^*. A *linguagem vazia* $\emptyset$ não contém nenhuma string e não deve ser confundida com $\{\varepsilon\}$, que contém apenas uma string.

adicionalmente, $\mathcal{L} \subset \Sigma^*$ é dita ser *regular* se obedece a um dos casos (SIPSER, 2006):

- $\mathcal{L} = \emptyset$.
- $\mathcal{L} = \{\varepsilon\}$.
- $\mathcal{L} = \{c\}$, onde $c \in \Sigma$.
- $\mathcal{L}$ pode ser construída a partir das seguintes operações:
  - A *concatenação* de duas linguagens regulares L_1 e L_2 sob um alfabeto Σ é denotada por:

$$L_1 \circ L_2 = \{wx \in \Sigma^* \mid w \in L_1 \wedge x \in L_2\}$$

a notação L^n, onde $n \geq 0$, representa o seguinte conjunto:

$$L^n = \underbrace{L \circ L \circ \cdots \circ L \circ L}_{\text{n vezes}} = \{w_0 w_1 \dots w_{n-1} w_n \in \Sigma^* \mid \forall w_i \in L\}$$

também é possível omitir a operação "$\circ$", deixando-a implícita, por exemplo:

$$L_1 \circ L_2 \equiv L_1 L_2$$

– A *união* de duas linguagens regulares L_1 e L_2 sob um alfabeto Σ é denotada por:

$$L_1 \cup L_2 = \{y \in \Sigma^* \mid y \in L_1 \vee y \in L_2\}$$

– O *fecho de Kleene* de uma linguagem regular L, denotado por L^*, representa a seguinte união infinita:

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

Na aritmética, podemos usar as operações $+$ e $\times$ para definir expressões como "$(5+3)\times 4$" que, quando computada, tem valor 32. Similarmente podemos usar o formalismo de Kleene para definir expressões que descrevem linguagens. Expressões regulares são formas compactas de escrever linguagens regulares. Por exemplo, a expressão "$(0 \cup 1)0^*$" descreve uma linguagem regular $\mathcal{L}$ tal que:

$$\mathcal{L} = \{0, 1, 00, 10, 000, 100, \dots\}$$

que contém todos as strings de Σ que possuem uma quantidade igual de zeros e uns.

2.4 Álgebra Abstrata

Esta seção é uma adaptação de (PIN, 2010) e visa definir os *semianéis*, que são usados no algoritmo estudado no próximo capítulo.

2.4.1 Magmas, Semigrupos e Monóides

Os Magmas (ou Grupóides) são estruturas algébricas simples (e pouco divulgadas). Um Magma é definido como um conjunto X equipado com uma operação binária $\oplus : X \times X \to X$ fechada (em X). Como exemplos de Magmas pode-se citar:

- O conjunto dos Naturais munidos da operação de adição, $(\mathbb{N}, +)$, forma um Magma.
- Os Naturais munidos da operação de subtração, $(\mathbb{N}, -)$, também forma um Magma.
- $(\mathbb{N}, \min)$ também é um exemplo de um Magma.

Apesar de serem extremamente comuns, os magmas não são tão úteis quanto os *semigrupos* ou os *monóides*. Um semigrupo é um conjunto S dotado de uma operação binária fechada ($S \times S \to S$) e associativa, por conveniência chamada de multiplicação. Exemplos de semigrupos são:

- Os inteiros positivos ($\mathbb{Z}^+$) munidos da operação de adição: $(\mathbb{Z}^+, +)$.
- O conjunto com valores booleanos $\mathbb{B} = \{\top, \bot\}$ munido com as operações de disjunção ou conjunção: $(\mathbb{B}, \vee)$ ou $(\mathbb{B}, \wedge)$.
- O conjunto das `strings` dotado de uma operação de concatenação (`+`) também forma um semigrupo:

$$\texttt{"hello"} + (\texttt{", "} + \texttt{"world"}) = (\texttt{"hello"} + \texttt{", "}) + \texttt{"world"}$$

Um monóide é uma estrutura $(X, \oplus, \bar{1})$ onde $(X, \oplus)$ é um semigrupo contendo um elemento neutro $\bar{1} \in X$ com a seguinte propriedade:

$$\forall x \in X, \bar{1} \oplus x = x \oplus \bar{1} = x$$

- O conjunto dos números naturais munidos da operação de adição ou de multiplicação tendo, respectivamente, 0 e 1 como elementos neutros forman os seguintes monóides: $(\mathbb{N}, +, 0)$ e $(\mathbb{N}, \cdot, 1)$.
- As `strings` dotadas de uma operação de concatenação e tendo a `string` vazia ε como elemento neutro também formam um monóide.

$$\texttt{s} + \varepsilon = \varepsilon + s = s$$

2.4.2 Semianéis

Um semianel é uma estrutura consistindo de um conjunto X, duas operações binárias $\oplus, \otimes : X \times X \to X$ e dois elementos neutros $\bar{0}, \bar{1}$. As estruturas $(X, \oplus, \bar{0})$ e $(X, \otimes, \bar{1})$ também são monóides e devem obedecer as seguintes restrições:

- Comutatividade da adição:

$$a \oplus b = b \oplus a$$

- O elemento $\bar{0}$ aniquila a multiplicação:

$$a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$$

- Distributividade da multiplicação e adição:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$
$$(a \oplus b) \otimes c = (a \otimes b) \oplus (a \otimes c)$$

2.5 Testes baseados em propriedades

John Hughes e Koes Clessen descrevem em (CLAESSEN; HUGHES, 2011) que o custo para escrita de testes motiva esforços para automatizá-los. O resultado de seu trabalho conjunto foi o QuickCheck, uma biblioteca em Haskell para geração de testes aleatórios a partir de propriedades.

Neste tipo de ferramenta o programador deve definir a especificação de um programa, através da criação de propriedades que devem ser satisfeitas (usando combinadores disponibilizados pela biblioteca), que são então usados pelo QuickCheck para fazer para gerar analisar a distribuição dos dados, gerar automaticamente testes e, quando encontrado um contra-exemplo à propriedade, minimizá-lo de forma a encontrar um contra-exemplo mínimo (e mais fácil de ser entendido).

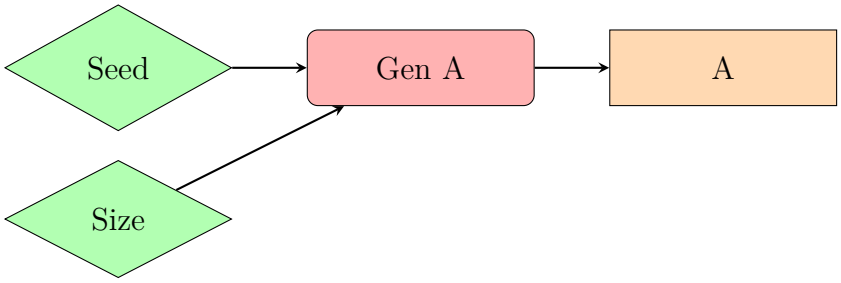


Figura 6 – Estrutura de um Gerador `Gen A`, que gera elementos do tipo `A`.

Ao lidarmos com testes baseados em propriedades não podemos assumir que receberemos tudo de graça. O problema agora passa da criação e escrita de bons casos de testes para a preocupação com o desenvolvimentos de geradores e boas propriedades que devem ser obedecidas. Geradores são primitivas oferecidas pela biblioteca QuickChick e podem ser compostos de forma gerar estruturas comuns (como listas de tipos de primitivos) ou customizadas (estruturas definidas pelo próprio usuário).

Como exemplo, tome as seguintes primitivas: `arbirtrary` e `elements`. A primeira tem como entrada um tipo A e retorna elementos arbirtrários deste tipo. A segunda tem como entrada uma lista, que define um intervalo para as saídas possíveis. A primitiva `sample` faz uma amostragem e toma um gerador como entrada.

```
*Main> import Test.QuickCheck
*Main Test.QuickCheck> :t sample
sample :: Show a => Gen a -> IO ()
*Main Test.QuickCheck> sample (arbitrary :: Gen Int)
0
0
-2
1
7
6
-7
0
-7
-10
-14
```

```
*Main Test.QuickCheck> :t elements
elements :: [a] -> Gen a
*Main Test.QuickCheck> sample $ elements [1..100]
87
59
5
50
25
23
10
29
53
81
63
```

Listing 2 – Geração de inteiros arbitrários usando as primitivas `arbitrary` e `elements`.

Assim como em testes normais, é necessário que se tome cuidado com a eficiência de cada teste. O exemplo abaixo demonstra a utilizadação da primitiva `suchThat`, que toma gera elementos obedecendo um predicado. Utilizamos um exemplo extramamente ineficiente, pois inteiros aleatórios serão gerados e descartados se não obedecerem nosso predicado. O segundo exemplo usa o fato dos geradores serem instâncias de um `functor` ($\langle\$\rangle$), os inteiros não são mais descartados na aplicação da função ($\lambda x.(3000 + \texttt{abs x})$).

```
*Main Test.QuickCheck> :set +s
*Main Test.QuickCheck> sample \
                     (arbitrary `suchThat` (>=3000))
3001
3014
3002
3000
3003
3005
3009
3017
3007
3008
3002
(7.18 secs, 11,296,080,304 bytes)
```

```
*Main Test.QuickCheck> sample $ \
                              (\x -> (3000 + abs x)) <$> arbitrary
3000
3000
3001
3003
3007
3008
3012
3011
3001
3014
3015
(0.01 secs, 592,040 bytes)
```

Listing 3 – Comparação do desempenho da versão ingênua (esquerda) com a versão que utiliza `functors` (direita).

Devido ao sucesso do QuickCheck outras linguagens começaram a implementá-lo. Recentemente o Coq ganhou seu próprio framework para geração de testes baseados em propriedades, o QuickChick (DÉNÈS et al., 2014), que implementa o protocolo original do QuickCheck.

3 Regex Play em COQ

Antes de usar o provador automatico para um dado algoritmo é necessário representá-lo usando as abstrações da linguagem COQ. Para esse trabalho, vamos usar a *functional pearl* "A Play on Regular Expressions" (FISCHER; HUCH; WILKE, 2010). Nesse artigo, os autores apresentararam o algoritmo em três atos, cada qual subdividido em 2 ou 3 cenas. Cada cena descreve uma implementação diferente para expressões regulares, que aumenta, gradualmente em nível de complexidade e generalidade.

O primeiro ato divide-se em duas cenas, a primeira busca descrever os `regexes` como `ADTs`, a cena seguinte trata de adicionar pesos à estrutura desenvolvida na primeira parte via `semirings`. O segundo ato busca implementar o algoritmo de Glushkov para `matchings` mais eficientes e adiciona a funcionalidade de computar o maior matching à esquerda. O último ato adiciona `lazyness` na versão do ato anterior. Ao final do artigo original de (FISCHER; HUCH; WILKE, 2010) tem-se uma implementação elegante em Haskell. O algoritmo em questão não apenas conseguia competir em termos de eficiência com implementações profissionais em `C++`, mas também era mais genérico, podendo resolver problemas como a contagem de `matchings` de uma `string` com um `regex` ou capturar o menor `matching` à esquerda.

3.1 Implementação do primeiro ato

O artigo começa com uma implementação básica para expressões regulares, que pode feita através de uma ADT. Dado um regex r, dizemos que uma string s é aceita por r nos seguintes casos:

```
data Reg                    ghci> let nocs = Reg (Alt (Sym 'a') (Sym 'b'))
  = Eps                     ghci> let onec = Seq nocs (Sym 'c')
  | Sym Char                ghci> let evencs = Seq (Rep (Seq onec onec)) nocs
  | Alt Reg                 ghci> accept evencs "cc"
        Reg                 True
  | Seq Reg                 ghci> accept evencs "ccabab"
        Reg                 True
  | Rep Reg
```

Listing 4 – Regex simples em Haskell.

- Se $r = \varepsilon$, então $c = \varepsilon$.

- Se $s = c$, onde c é um caratere, então $s = c$.
- Se $r = (p \mid q)$, onde p e q são `regexes`, a união é aceita se pelo menos uma das partes é aceita por r.
- Se $r = (pq)$, onde p e q são `regexes`, a concatenação é aceita se ambas as partes são aceitas por r.
- Se $r = u^*$, o fecho de Kleene é aceito se s é aceito zero ou mais vezes.

```
accept :: Reg -> String -> Bool
accept Eps u       = null u
accept (Sym c) u   = u == [c]
accept (Alt p q) u = accept p u || accept q u
accept (Seq p q) u = or [accept p u1 && accept q u2 | (u1, u2) <- split u]
accept (Rep r) u   = or [and [accept r u_i | u_i <- ps] | ps <- parts u]
```

Listing 5 – Matching simples em Haskell.

as funções `split` e `parts` podem ser consultadas no Apêndice A. Na segunda cena, a adição de `semirings` à estrutura original permite que não só se tenha um valor booleano como resposta (caso uma string tenha `match` ou não), mas a quantidade de `matchings` que ocorrem (valor inteiro).

```
data RegW c s                ghci> let as = Alt (Sym 'a') (Rep (Sym 'a'))
  = EpsW                     ghci> acceptW (weighted as) "a" :: Int
  | SymW (c -> s)            2
  | AltW (RegW c s)          ghci> let bs = Alt (Sym 'b') (Rep (Sym 'b'))
         (RegW c s)          ghci> acceptW (weighted (Seq as bs)) "ab" :: Int
  | SeqW (RegW c s)          4
         (RegW c s)          ghci> acceptW (weighted (Seq as bs)) "ab" :: Bool
  | RepW (RegW c s)          True
```

Listing 6 – Regex com pesos em Haskell.

A versão anterior pode ser alterada com duas pequenas mudanças nos tipos báscisp (`Eps` e `Sym`). O nova versão de `Sym` deverá tomar um predicado que compara dois símbolos e retorna o $\bar{0}$ ou $\bar{1}$ de um `semiring`. Não apenas isso, deve-se implementar um método que transforme um `regex` normal numa versão com pesos.

```haskell
class Semiring a where
  zero, one :: a
  (<.>) :: a -> a -> a
  (<+>) :: a -> a -> a

infixl 7 <.>

infixl 6 <+>

sym :: Semiring s => Char -> RegW Char s
sym c =
  SymW
    (\x ->
       if x == c
         then one
         else zero)

weighted :: Semiring s => Reg -> RegW Char s
weighted Eps       = EpsW
weighted (Sym c)   = sym c
weighted (Alt p q) = AltW (weighted p) (weighted p)
weighted (Seq p q) = SeqW (weighted p) (weighted q)
weighted (Rep r)   = RepW (weighted r)
```

Listing 7 – Semirings e conversão de um Regex normal em Haskell.

A nova versão da função `accept` mantém a mesma forma da definição original. Contudo, os operadores agora são mais genéricos, pois fazem operações em `semirings`.

```haskell
acceptW :: Semiring s => RegW c s -> [c] -> s
acceptW EpsW u =
  if null u
    then one
    else zero
acceptW (SymW f) u =
  case u of
    [c] -> f c
    _   -> zero
acceptW (AltW p q) u = acceptW p u <+> acceptW q u
acceptW (SeqW p q) u =
  sum' [acceptW p u1 <.> acceptW q u2 | (u1, u2) <- split u]
acceptW (RepW r) u = sum' [prod [acceptW r ui | ui <- ps] | ps <- parts u]

sum' :: Semiring s => [s] -> s
sum' = foldr (<+>) zero

prod :: Semiring s => [s] -> s
prod = foldr (<.>) one
```

Listing 8 – Accept com pesos em Haskell.

3.1.1 Expressões regulares em COQ

A representação de um regex simples é similar à utilização de ADTs em Haskell, contudo em COQ é possível utilizar Tipos Indutivos, que são uma generalização dos ADTs. A palavra vazia é representada por `Eps`, um símbolo c (onde c é tipo `ascii`) é representado por uma função `Sym c` que toma como argumento um caractere e retorna um regex. Combinações `Alt` e `Seq` são codificadas como funções que tomam dois argumentos (do tipo regex) e retornam um regex, o fecho de Kleene é uma função de um argumento que recebe e retorna um regex.

```
Inductive regex : Type :=
  | Eps : regex
  | Sym : ascii -> regex
  | Alt : regex -> regex -> regex
  | Seq : regex -> regex -> regex
```

Listing 9 – Definição indutiva de uma expressão regular simples.

Como COQ não possui compreensão de listas por padrão, as funções polimórificas `split` e `parts` foram implementadas usando funções de ordem superior. Definições auxiliares (como `non_empty` e `athead`) podem ser consultadas no Anexo A.

```
Fixpoint split {X : Type} (l : list X) : list (list X * list X) :=
  match l with
    | [] => [([], [])]
    | c::cs =>
      ([], c::cs) :: map (fun '(s1, s2) => (c :: s1, s2)) (split cs)
  end.


Fixpoint parts {X : Type}
  (xs : list X) : list (list (list X)) :=
  match xs with
    | [] => [[]]
    | [c] => [[[c]]]
    | (c :: cs) =>
      flat_map (fun ps => [ athead c ps; [c] :: ps])
               (filter non_empty (parts cs))
```

Listing 10 – Funções auxiliares para corte e partição de listas polimórficas.

A definição do matching em expressões regulares simples também é similar à solução em Haskell. Utilizando, novamente, high order functions ao invés de compreensão de listas. Definições extras usadas na funções podem ser consultadas no Anexo B.

```
Fixpoint accept (r : regex) (u : lstring) : bool :=
  match r with
    | Eps => is_null u
    | Sym c => eq_lstring [c] u
    | Alt p q => accept p u || accept q u
    | Seq p q =>
      or_fold (map (fun '(u1, u2) => accept p u1 && accept q u2)
                   (split u))
    | Rep x =>
      let mparts r' m :=
          map (fun x =>  accept r' x) m in
      let nparts r' m :=
          map (fun x => mparts r' x) m in
      let result r' m :=
          or_fold (map (and_fold)
                       (nparts r' (parts m)))
      in
      result x u
```

3.1.2 Semirings

Diferentemente da implementação em Haskell, o sistema de tipos usado em COQ força que algo seja uma instância de Semiring apenas se satifazer todas as restrições, confirmadas através de provas. As provas de que os tipos Booleanos e Naturais como Semirings são triviais e podem ser consultadas no Anexo C.

```
Require Import Coq.Bool.Bool Coq.Arith.PeanoNat.

Class Semiring {X : Type}
  (zero : X) (one : X)
  (add : X -> X -> X) (mul : X -> X -> X) :=
  {
    (* Semiring rules *)

    add_comm : forall (x y : X), add x y = add y x;

    zero_mul_r : forall (x : X), mul x zero = zero;

    zero_mul_l : forall (x : X), mul zero x = zero;
```

```
  distr_mul : forall (x y z : X),
      mul x (add y z) = add (mul x y) (mul x z);

  distr_add : forall (x y z : X),
      mul (add x y) z = add (mul x z) (mul y z);

  (* Monoid rules*)
  add_associativity : forall (x y z : X),
      add x (add y z) = add (add x y) z;

  add_zero_l : forall (x : X), add zero x = x;
  add_zero_r : forall (x : X), add x zero = x;

  mul_associativity : forall (x y z : X),
      mul x (mul y z) = mul (mul x y) z;

  mul_one_l : forall (x : X), mul one x = x;
  mul_one_r : forall (x : X), mul x one = x;
}.
```

4 Testes Baseados em Proprieades

4.1 Geradores

Uma das vantagens de bibliotecas como o QuickChick é que dificilmente precisamos nos preocupar com a codificação de bons casos de testes. A responsabilidade do desenvolvedor agora é transferida para a criação de bons geradores e especificação de boas propriedades, utilizando as primitivas já fornecidas pela biblioteca.

Para que se utilize a biblioteca QuickChick é necessário, primeiro, que se garanta que as estruturas que pretendemos gerar são instâncias da typeclass `Show`. Uma estrutura é membro de `Show` se implementa uma função `show`, que especifica como se deve representar uma estrutura como uma `string`. Temos abaixo uma possível definição de `show_regex`, que torna a estrutura definida no capítulo anterior uma instância da typeclass `Show`.

```
Fixpoint show_regex (r : regex) : string :=
  match r with
    | Eps ⇒ "ε"
    | Sym c ⇒ show_ascii c
    | Alt p q ⇒
      "(" ++ (show_regex p) ++ "|" ++ (show_regex q) ++ ")"
    | Seq p q ⇒
      "(" ++ (show_regex p) ++ (show_regex q) ++ ")"
    | Rep x ⇒
      show_regex x ++ "*"
  end.

Instance ShowRegex : Show regex :=
  {
    show := show_regex
  }.
```

Figura 7 – Regexes como instâncias de Show

Os geradores para caracteres `ascii` e `strings` podem ser definidos usando primtivas simples. A geração de caracteres pode ser feita via `choose`, que toma um intervalo como entrada e retorna números naturais, que são convertidos para caracteres com a função `ascii_of_nat`.

Em COQ, `strings` não são definidas como listas de caracteres. O gerador `genString` retorna `strings` de tamanho n da biblioteca padrão e recebe como entrada um gerador de caracteres. O gerador `genLString` retorna listas de caracteres e recebe os mesmos argumentos.

```
Definition genAscii : G ascii :=
  (liftM ascii_of_nat (choose (97,122))).
```

```
Fixpoint genString (n : nat) (g1 : G ascii)
  : G string :=
  match n with
    | 0 => ret EmptyString
    | S m => liftM2 String g1 (genString m g1)
  end.

Definition genLString (n : nat) (g : G ascii) :=
  vectorOf n g.
```

Em estruturas mais complexas e recursivas (como árvores) é necessário garantir que o gerador irá terminar. Sendo assim, é comum passar a profundidade da árvore como argumento. Os `regexes` definidos no capítulo anterior são, basicamente, árvores. Para gerar `regexes` aleatórios utilizamos a primitiva `freq`, que toma como argumentos um gerador e uma lista de tuplas. O combinador `freq` escolhe um dos geradores da lista, o primeiro elemento de cada tupla representa um peso para a escolha do gerador. O valor 1 em `(1, ret Eps)` significa que geraremos strings vazias com uma probabilidade menor que os outros casos.

```
Fixpoint genRegex (n : nat) (g : G ascii)
  : G regex :=
  match n with
    | 0 => liftM Sym g
    | S m =>
      freq [
          (1, ret Eps);
          (3, liftM Sym g);
          (4, liftM2 Alt (genRegex m g) (genRegex m g));
          (4, liftM2 Seq (genRegex m g) (genRegex m g));
          (3, liftM Rep (genRegex m g))
      ]
  end.
```

```
QuickChecking (genRegex 5 genAscii)
[((n|(((ts)|ε)o*))|((k|k)(ev))**); (ε((m|(d|(t|e)))|((g*|(uc))|o*))); (((ε|((yi)(t|p)))|(i|i**))|v); (((r((bw)|(f|q)))|m)|z**); ((((
dg)|(p|b))((we)m))*|ε); ((x|z*)**((k|(qa)|(rv))((j|a)*((lg)k)))); (((gg)*(x*c*))ε)*; o; ((((g|a*)ε)|((ε|(ck))j*))|((w|(jy))|((so)|
(pe)))*); (((k*|ε)|((q*y*)x))(k(((ue)ε)(uf)))); (mq)]
```

Figura 8 – Exemplo da saída de `genRegex 5 genAscii`.

5 Conclusão

Testes são essenciais ao desenvolvimento de software e hardware. Atualmente a maioria das linguagens de programação possuem *frameworks* para testes unitários e de integração. Porém, os testes servem apenas para demonstrar a presença e nunca a ausência. Para isso, seria necessários provas matemáticas.

Este trabalho explorou algumas ferramentas e metodologias utilizadas para provar a corretude matemática da especificação de programas. Começou-se a partir dos problemas filosóficos da matemática que acidentalmente levaram a criação a ciência da computação. Além da criação dos formalismos que demonstram como programas escritos em alguma versão do λ-Cálculo são equivalentes a provas da Lógica.

A formalização e prova matemática de um algoritmo demanda um grande esforço, mesmo usando um provador automático. Alternativamente, testes baseados em propriedades se apresentam com uma solução intermediária entre os testes unitários e as provas de corretude. Pois diferente dos testes de unidade, os testes baseado em propriedades geram casos de testes automatizados e aleatórios. Porém, vale salientar que os testes baseados em propriedades não garantem a corretude de um sistema, e dependem da criação de boas propriedades.

Este trabalho conseguiu então explorar alguns conceitos essências e básicos para garantir códigos mais robustos e com maior qualidade. Então, abre-se a oportunidade para continuidade e trabalhos futuros. Pode-se citar os seguintes trabalhos:

- A implementação de todos os atos do artigo original de (FISCHER; HUCH; WILKE, 2010).
- Melhorar os geradores já implementados.
- Definir conjecturas (propriedades) e verificar sua corretude usando QuickChick.
- Utilizar as conjecturas que passam pelos testes baseados em propriedades para derivar provas formais.
- Fazer a extração em Haskell das implementações feitas em COQ.

Referências

BAEZ, J.; STAY, M. Physics, topology, logic and computation: a rosetta stone. In: *New structures for physics*. [S.l.]: Springer, 2010. p. 95–172. Citado na página 31.

BRUIJN, N. G. de. Automath, a language for mathematics. In: *Automation of Reasoning*. [S.l.]: Springer, 1983. p. 159–200. Citado na página 33.

CHEN, H. et al. Verifying a high-performance crash-safe file system using a tree specification. In: ACM. *Proceedings of the 26th Symposium on Operating Systems Principles*. [S.l.], 2017. p. 270–286. Citado na página 26.

CLAESSEN, K.; HUGHES, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, ACM, v. 46, n. 4, p. 53–64, 2011. Citado 2 vezes nas páginas 26 e 42.

COQUAND, T.; HUET, G. *The calculus of constructions*. Tese (Doutorado) — INRIA, 1986. Citado na página 26.

DÉNÈS, M. et al. Quickchick: Property-based testing for coq. In: *The Coq Workshop*. [S.l.: s.n.], 2014. Citado na página 43.

DOOLEY, J. *Software development and professional practice*. [S.l.]: Apress, 2011. Citado na página 25.

FISCHER, S.; HUCH, F.; WILKE, T. A play on regular expressions: functional pearl. In: ACM. *ACM Sigplan Notices*. [S.l.], 2010. v. 45, n. 9, p. 357–368. Citado 2 vezes nas páginas 45 e 53.

GONTHIER, G. Formal proof–the four-color theorem. *Notices of the AMS*, v. 55, n. 11, p. 1382–1393, 2008. Citado na página 26.

GU, L. et al. Certikos: a certified kernel for secure cloud computing. In: ACM. *Proceedings of the Second Asia-Pacific Workshop on Systems*. [S.l.], 2011. p. 3. Citado na página 26.

HINDLEY, J. R.; SELDIN, J. P. *Lambda-calculus and Combinators, an Introduction*. [S.l.]: Cambridge University Press Cambridge, 2008. v. 13. Citado 2 vezes nas páginas 31 e 32.

IRVINE, A. D.; DEUTSCH, H. Russell's paradox. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Winter 2016. [S.l.]: Metaphysics Research Lab, Stanford University, 2016. Citado 2 vezes nas páginas 29 e 35.

KLEENE, S. C. *Representation of events in nerve nets and finite automata*. [S.l.], 1951. Citado na página 36.

KLEENE, S. C.; ROSSER, J. B. The inconsistency of certain formal logics. *Annals of Mathematics*, JSTOR, p. 630–636, 1935. Citado na página 32.

LEROY, X. et al. The compcert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, v. 53, 2012. Citado na página 26.

MANNA, Z.; WALDINGER, R. Logical basis for computer programming. 1985. Citado na página 35.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, Springer, v. 5, n. 4, p. 115–133, 1943. Citado na página 35.

NEMETH, E. *UNIX and Linux System Administration Handbook, 5/e*. [S.l.]: Addison-Wesley Professional, 2017. Citado na página 37.

PIN, J.-É. Mathematical foundations of automata theory. *Lecture notes LIAFA, Université Paris*, v. 7, 2010. Citado na página 40.

RANDELL, B. The 1968/69 nato software engineering reports. *History of Software Engineering*, Citeseer, v. 37, 1996. Citado na página 25.

SHAPIRO, L.; BEALL, J. Curry's paradox. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Summer 2018. [S.l.]: Metaphysics Research Lab, Stanford University, 2018. Citado na página 32.

SIPSER, M. *Introduction to the Theory of Computation*. [S.l.]: Thomson Course Technology Boston, 2006. v. 2. Citado na página 39.

TEAM, T. C. D. *The Coq Proof Assistant, version 8.9.0*. 2019. Disponível em: <https://doi.org/10.5281/zenodo.2554024>. Citado na página 26.

THOMPSON, K. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, ACM, v. 11, n. 6, p. 419–422, 1968. Citado na página 36.

TROELSTRA, A. S. Constructivism and proof theory. Citeseer, 2003. Citado na página 30.

VOEVODSKY, V. et al. *UniMath — a computer-checked library of univalent mathematics*. 2014. available at <https://github.com/UniMath/UniMath>. Disponível em: <https://github.com/UniMath/UniMath>. Citado na página 26.

WADLER, P. Proofs are programs: 19th century logic and 21st century computing. 2000. Citado 3 vezes nas páginas 29, 31 e 35.

WADLER, P. Propositions as types. *Commun. ACM*, v. 58, n. 12, p. 75–84, 2015. Citado 2 vezes nas páginas 30 e 33.

ZALTA, E. N. Gottlob frege. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Summer 2018. [S.l.]: Metaphysics Research Lab, Stanford University, 2018. Citado na página 29.

Anexos

ANEXO A – Utils

Definições auxiliares em Haskell:

```
-- Decomposes a list into pairs
split :: [a] -> [([a], [a])]
split []     = [([], [])]
split (c:cs) = ([], c : cs) : [(c : s1, s2) | (s1, s2) <- split cs]

-- Partitions a given list
parts :: [a] -> [[[a]]]
parts []     = [[]]
parts [c]    = [[[c]]]
parts (c:cs) = concat [[(c : p) : ps, [c] : p : ps] | p:ps <- parts cs]
```

Definições auxiliares para as HOF em Coq:

```
Definition athead {X : Type}
  (x : X) (xs : list (list X)) :=
  match xs with
    | [] => []
    | (y :: ys) => (x :: y) :: ys
  end.

Definition non_empty {X : Type} (xs : list (list X)) :=
  match xs with
    | [] => false
    | _ => true
  end.
```

ANEXO B – Accept

B.1 CStrings.v

Algumas definições extras para manipular strings como listas de `ascii`.

```
Definition lstring := list ascii.

Definition eq_ascii (a1 a2 : ascii) :=
  match a1, a2 with
  | Ascii b1 b2 b3 b4 b5 b6 b7 b8, Ascii c1 c2 c3 c4 c5 c6 c7 c8 =>
    (eqb b1 c1) && (eqb b2 c2) && (eqb b3 c3) && (eqb b4 c4) &&
    (eqb b5 c5) && (eqb b6 c6) && (eqb b7 c7) && (eqb b8 c8)
  end.


Fixpoint eq_string (s1 s2 : string) :=
  match s1, s2 with
    | EmptyString, EmptyString => true
    | _, EmptyString => false
    | EmptyString, _ => false
    | String x xs, String y ys => (eq_ascii x y) && eq_string xs ys
  end.


Fixpoint eq_lstring (s1 s2 : lstring) :=
  match s1, s2 with
    | [], [] => true
    | _, [] => false
    | [] , _ => false
    | x::xs, y::ys => (eq_ascii x y) && eq_lstring xs ys
  end.
```

B.2 Accept.v

Definições extras para a função `accept` em COQ.

```
Definition is_null (s : lstring) : bool :=
  match s with
    | [] => true
    | _  => false
  end.

Definition or_fold (l : list bool) : bool :=
  fold_right (orb) false l.

Definition and_fold (l : list bool) : bool :=
  fold_right (andb) true l.
```

ANEXO C – Instâncias de Semirings

C.1 Semiring.hs

```haskell
instance Semiring Bool where
  zero = False
  one = True
  (<+>) = (||)
  (<.>) = (&&)

instance Semiring Int where
  zero = 0
  one = 1
  (<+>) = (+)
  (<.>) = (*)
```

C.2 Semiring.v

```
Instance BoolSemiring :
  Semiring false true (orb) (andb) :=
  {}.
  (* add_comm *)
  Proof.
    intros. apply orb_comm.
  (* zero_mul_r *)
  Proof.
    intros. apply andb_false_r.
  (* zero_mul_l *)
  Proof.
    intros. apply andb_false_l.
  (* distr_mul *)
  Proof.
    intros. apply andb_orb_distrib_r.
  (* distr_add *)
  Proof.
    intros. apply andb_orb_distrib_l.
  (* Monoid proofs *)
```

```
  (* add_associativity *)
  Proof.
    intros. apply orb_assoc.
  (* add_zero_l *)
  Proof.
    intros. apply orb_false_l.
  (* add_zero_r *)
  Proof.
    intros. apply orb_false_r.
  (* mul_associativity *)
  Proof.
    intros. apply andb_assoc.
  (* add_zero_l *)
  Proof.
    intros. apply andb_true_l.
  (* add_zero_r *)
  Proof.
    intros. apply andb_true_r.
  Defined.



Instance NatSemiring : Semiring
  0 1 Nat.add Nat.mul :=
  {}.
  (* add_comm *)
  Proof.
    intros. apply Nat.add_comm.
  (* zero_mul_r *)
  Proof.
    intros. apply Nat.mul_0_r.
  (* zero_mul_l *)
  Proof.
    intros. apply Nat.mul_0_l.
  (* distr_mul *)
  Proof.
    intros. apply Nat.mul_add_distr_l.
  (* distr_add *)
  Proof.
    intros. apply Nat.mul_add_distr_r.
```

```
(* Monoid proofs *)
(* add_associativity *)
Proof.
  intros. apply Nat.add_assoc.
(* add_zero_l *)
Proof.
  intros. apply Nat.add_0_l.
(* add_zero_r *)
Proof.
  intros. apply Nat.add_0_r.
(* mul_associativity *)
Proof.
  intros. apply Nat.mul_assoc.
(* add_zero_l *)
Proof.
  intros. apply Nat.mul_1_l.
(* add_zero_r *)
Proof.
  intros. apply Nat.mul_1_r.
Defined.
```