NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

CAROLINA PEREIRA DUARTE

Licenciada em Engenharia Informática

COMPOSIÇÃO DE MODELOS UTILIZANDO OPERAÇÕES REGULARES EM OCAMLFLAT/OFLAT

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Julho, 2024

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

COMPOSIÇÃO DE MODELOS UTILIZANDO OPERAÇÕES REGULARES EM OCAMLFLAT/OFLAT

CAROLINA PEREIRA DUARTE

Licenciada em Engenharia Informática

Orientador: Artur Miguel Dias

Professor Auxiliar, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia

Júri

Presidente: Susana Maria dos Santos Nascimento Martins de Almeida
Professora Auxiliar, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia

Arguente: Rogério Ventura Lages dos Santos Reis
Professor Auxiliar, Faculdade de Ciências da Universidade do Porto.

Vogal: Artur Miguel Dias
Professor Auxiliar, Universidade NOVA de Lisboa - Faculdade de Ciências e Tecnologia

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Julho, 2024

Composição de modelos utilizando operações regulares em OCamlFLAT/O-FLAT

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v6.10.10) [12].

Agradecimentos

Gostaria de agradecer ao meu orientador, Artur Miguel Dias, pela disponibilidade e apoio dado ao longo de toda a dissertação. E à minha família, amigos e namorado que estiveram ao meu lado durante esta fase.

Resumo

A teoria FLAT (Formal Languages and Automata Theory) é uma área importantíssima, que corresponde a parte dos fundamentos da Informática. No entanto, a sua natureza matemática e formal nem sempre permite aos alunos entenderem por completo os seus diferentes conceitos. Para apoiar o ensino desta teoria têm sido desenvolvidas múltiplas ferramentas informáticas que tentam ajudar os alunos de forma visual e intuitiva.

Duas ferramentas desenvolvidas na FCT/UNL são a biblioteca OCamlFLAT e a aplicação Web OFLAT. Ambas as ferramentas foram escritas em OCaml, tentando usar deliberadamente um estilo funcional declarativo, que em alguns casos se consegue aproximar dos formalismos teóricos ensinados aos alunos. As duas ferramentas encontram-se em evolução e crescimento.

Esta dissertação adicionou uma funcionalidade que é praticamente ausente nas ferramentas existentes: a composição de modelos. Foi dada primazia à composição usando as operações regulares (união, a concatenação e o fecho de Kleene), mas também foi considerada a operação de interseção, para introduzir um exemplo de operação não regular.

A intervenção ocorreu ao nível da biblioteca, com a introdução de novas representações e novas operações (por exemplo, o cálculo de um autómato finito a partir de uma composição de autómatos finitos), e ao nível da aplicação gráfica, com a criação de uma interface pedagógica para lidar com a composição de modelos.

Palavras-chave: Teoria da Computação, Composição de Modelos, Interface gráfica pedagógica

Abstract

FLAT theory (Formal Languages and Automata Theory) is a very important area, which corresponds to part of the foundations of Information Technology. However, its mathematical and formal nature does not always allow students to fully understand its different concepts. To support the teaching of this theory, multiple computer tools have been developed that try to help students in a visual and intuitive way.

Two tools developed at FCT/UNL are the OCamlFLAT library and the OFLAT web application. Both tools were written in OCaml, trying to deliberately use a declarative functional style, which in some cases comes close to the theoretical formalisms taught to students. Both tools are evolving and growing.

This dissertation added a functionality that is practically absent in existing tools: model composition. Priority was given to composition using the regular operations (union, concatenation and Kleene closure), but the intersection operation was also considered, to introduce an example of a non-regular operation.

The intervention took place at the library level, with the introduction of new representations and new operations (for example, the calculation of a finite automaton from a composition of finite automata), and at the graphical application level, with the creation of a pedagogical interface to deal with the composition of models.

Keywords: Computation theory, Composition of models, Automaton, Turing machines,

Índice

Índice de Figuras

Índice de Listagens

1

Introdução

1.1 Contexto

O estudo da FLAT (Formal Languages and Automata Theory) é frequente em cursos de Engenharia Informática e Ciência da Computação. No entanto, devido à sua natureza matemática e formal, os alunos muitas vezes têm dificuldade a entender o funcionamento dos modelos que caracterizam essa área de estudo. É por isso importante haver uma ferramenta que possa servir de complemento ao estudo da teoria FLAT. O ideal é ser prático e simples de usar e apresentar de maneira visual os modelos FLAT.

Ao longo dos últimos anos, tem sido desenvolvida na FCT-UNL a ferramenta OCaml-FLAT/OFLAT, bastante completa nesta área. Apesar de já se encontrar consideravelmente avançada, com suporte a vários modelos importantes, existem ainda várias funcionalidades que seria desejável adicionar.

Assim esta dissertação procura adicionar uma nova funcionalidade ao OCamlFLAT/OFLAT tanto ao nível de biblioteca OCamlFLAT como ao nível da interface gráfica OFLAT.

1.2 Objetivos

Esta dissertação visou a adição de suporte para a composição de modelos. Foi dada primazia à composição usando as operações regulares (união, a concatenação e o fecho de Kleene), mas também foi considerada a operação de interseção. De notar, que este tipo de funcionalidade é praticamente inexistente nas ferramentas FLAT atualmente existentes. Foi necessário estender a biblioteca OCamlFLAT para suportar a composição de modelos, e também foi necessário fazer o tratamento gráfico para ser possível criar e editar modelos compostos na plataforma OFLAT.

1.3 Estrutura do documento

Este documento encontra-se dividido em cinco capítulos da seguinte forma:

- **Introdução** — Introduz o tema desta dissertação, estando dividido em contexto, objetivos e estrutura do documento.
- **Teoria da Computação** — Dá a conhecer os conceitos teóricos dos modelos FLAT. Apresenta um subcapítulo que introduz a hierarquia de Chomsky seguido de quatro subcapítulos destinados aos quatro tipos de linguagem 3, 2, 1 e 0. Apresenta ainda um subcapítulo sobre composição de modelos e outro sobre problemas de decisão.
- **Ferramentas de aprendizagem sobre teoria FLAT** — Este capítulo dá a conhecer algumas ferramentas pedagógicas já existentes sobre teoria FLAT. As ferramentas mencionadas são o JFLAP, FAdo e FAT.
- **Plataforma OFLAT e OCamlFLAT** — Este capítulo apresenta a biblioteca OCamlFLAT e a aplicação OFLAT.
- **Tecnologias** — Discute várias questões técnicas que estão presentes na implementação gráfica.
- **Suporte para composição de modelos na biblioteca OCamlFLAT** — Este capítulo discute a implementação da composição de modelos na biblioteca OCamlFLAT, mais concretamente o módulo novo adicionado.
- **Suporte para composição de modelos na plataforma OFLAT** — Este capítulo discute a implementação do suporte para a composição de modelos na plataforma OFLAT.
- **Avaliação e trabalho futuro** — Este capítulo apresenta: um subcapítulo que discute como foi feita avaliação desta dissertação, um subcapítulo que explora possível trabalho futuro e um subcapítulo que contém as conclusões finais.

2

Teoria da Computação

Em Teoria da Computação, estudam-se diversos modelos de computação com diferentes níveis de poder expressivo.

Para discutir e comparar diferentes níveis de poder expressivo, é inevitável considerar os algoritmos no contexto de algum domínio de aplicação. O domínio de aplicação tipicamente escolhido é o da sintaxe de linguagens formais.

Há duas técnicas de definição de linguagens que tipicamente se usam. A primeira técnica baseia-se no conceito de geração e usa o formalismo das gramáticas. Outra técnica baseia-se no mecanismo de reconhecimento e usa o formalismo dos autómatos.

2.1 Hierarquia de Chomsky

Noam Chomsky (com a ajuda de Marcel-Paul Schützenberger) identificou diversas classes de linguagens formais, com diferentes níveis de complicação de especificação, a requerer modelos de computação com diferentes níveis de poder expressivo para resolver o problema do reconhecimento dessas linguagens.

Existem quatro níveis: tipo 0, tipo 1, tipo 2 e tipo 3.

- tipo 0 — linguagens recursivamente enumeráveis
- tipo 1 — linguagens dependentes do contexto
- tipo 2 — linguagens independentes do contexto
- tipo 3 — linguagens regulares

A hierarquia de Chomsky envolve quatro níveis de tipos de linguagens. Cada tipo de linguagens tem associado um tipo de gerador e um tipo de reconhecedor.

2.2 Linguagens Regulares

Linguagens de nível 3 ou Linguagens regulares. Qualquer linguagem regular pode ser descrita por uma expressão regular. E é reconhecida por um autómato finito.

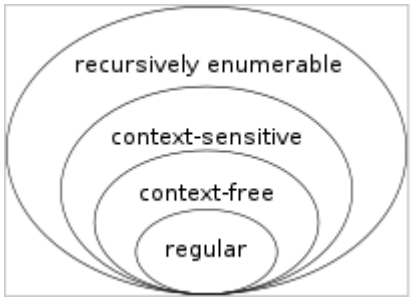

Figura 2.1: Hierarquia de Chomsky

2.2.1 Expressões Regulares

Uma **expressão regular** é uma forma de especificar uma linguagem regular [10]. Define-se por indução o conjunto de todas expressões regulares (sobre dado alfabeto) e a linguagem denotada por uma dada expressão [15].

Expressões regulares são definidas com base num alfabeto Σ (conjunto finito de símbolos) e um conjunto de operações. A expressão regular $\varnothing$ denota a linguagem vazia e a expressão regular ε denota a linguagem que contém apenas a palavra vazia. Para compor expressões regulares usam-se as três seguintes operações básicas: [15]

$$a, b, c \in \Sigma$$

(ab) — concatenação

$a + b$ — soma ou união

a^* — fecho de Kleene

Propriedades da concatenação: [13]

- Associativa $a(bc) = (ab)c$
- Elemento neutro $a\varepsilon = a$
- Elemento absorvente $a\varnothing = \varnothing$

Propriedades da soma:

- Associativa: $a + (b + c) = (a + b) + c$
- Elemento neutro: $a + \varnothing = \varnothing + a = a$
- Comutativa: $a + b = b + a$
- Idempotência: $a + a = a$

Propriedades de fecho de Kleene:

- $a^* + \varepsilon = a^*$

- $a^* + a = a^*$
- $(a + \varepsilon)^* = a^*$
- $(a^*)^* = a^*$
- $aa^* = a^*$
- $a(bc)^* = (ab)^*c$

Outras propriedades das expressões regulares:

- Distributiva à esquerda — $a(b + c) = ab + ac$
- Distributiva à direita — $(a + b)c = ac + bc$

Note-se que uma expressão regular define uma linguagem, no entanto uma linguagem pode ser definida por várias expressões regulares devido a estas propriedades.

Considerando o alfabeto anterior $\{a, b, c\}$ temos uma expressão regular $a(b+c)a^*$. Esta expressão gera palavras que começam com o símbolo a, seguido por b ou c, finalizando com zero ou mais a. Exemplos de palavras geradas por esta expressão são aba, $acaa$, ac. Devido à propriedade distributiva à esquerda, a expressão regular $a(b+c)a^*$ é equivalente à expressão $(ab + ac)a^*$, por isso ambas definem a mesma Linguagem Regular.

2.2.2 Autómatos Finitos

Uma Linguagem Regular é uma Linguagem reconhecida por um autómato finito.

Um autómato finito recebe como input uma string, e o output é apenas um de dois resultados: ou aceite ou rejeitado. Um autómato finito pode ser representado graficamente por um grafo, em que os nós etiquetados denotam os estados do autómato e os arcos etiquetados denotam transições [15].

Um autómato finito é definido da seguinte forma

$$A = \langle S, \Sigma, s, \delta, F \rangle$$

onde S é um conjunto finito de estados, Σ é o alfabeto da linguagem, $s \in S$ é o estado inicial, δ é a relação de transição em que $\delta \in S \times (\Sigma \cup \epsilon) \times S$, F é o conjunto de estados finais ou de aceitação [15].

Todos os autómatos têm um estado inicial onde o reconhecimento de uma palavra começa e podem ter vários estados finais.

Na figura 2.2 podemos ver o autómato finito que corresponde à expressão regular acima referida

A definição de autómato finito que aqui é considerada é a mesma usada na biblioteca OCamlFLAT. Trata-se da tradicional definição de autómato finito não determinista. Para um autómato em que a relação de transição seja uma função em vez de uma relação,

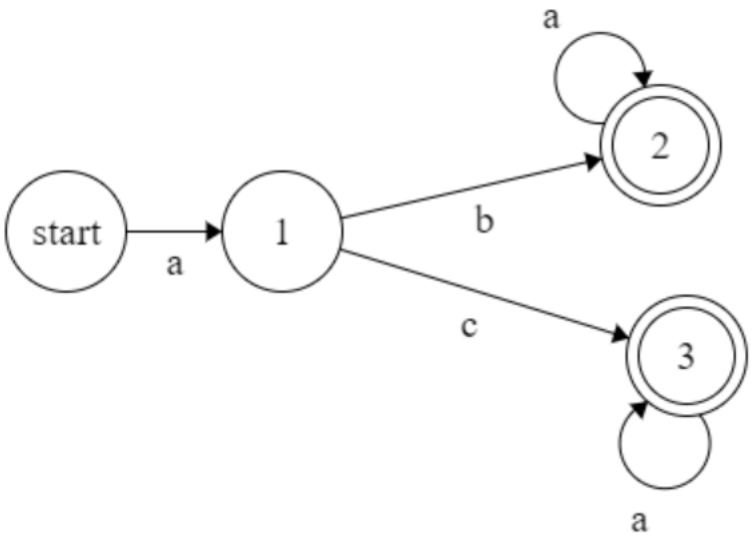

Figura 2.2: Autómato finito

esse autómato será determinista. Os autómatos não deterministas são geralmente mais fáceis de definir por um ser humano, mas os autómatos deterministas são mais simples de implementar numa máquina. Contudo, o poder computacional das duas variantes é idêntico.

2.3 Linguagens independentes de contexto

As linguagens independentes de contexto englobam as linguagens regulares, são um tipo de linguagem formal que pode ser definida através do uso de gramáticas livres de contexto. Estas gramáticas possuem regras de produção que permitem gerar sequências de símbolos de forma recursiva, envolvendo símbolos não-terminais e terminais. Têm várias aplicações na informática como, auxílio na implementação de compiladores e descrição de formatos de documentos.

2.3.1 Gramáticas livres de Contexto

Gramáticas Livres de contexto são dispositivos geradores das linguagens livres de contexto. O seu mecanismo de geração é a reescrita, que passa pela substituição de uma subpalavra por outra palavra, e é especificada por regras de produção. Uma Gramática independente de contexto é definida da seguinte forma [15]:

$$\langle V, T, P, S \rangle$$

Onde:

- V — É o conjunto(finito não vazio) de símbolos não terminais
- T — É o conjunto de símbolos terminais
- P — É o conjunto finito de produções, com $P \subseteq V \times (\ V \cup T)^*$
- S — É o símbolo inicial, com $S \in V$

Exemplo:

$$\langle \{P\}, \{a, b\}, \{\ (P, \varepsilon\ ),\ (P, aPa\ ),\ (P, bPb\ )\}, \{P\} \rangle$$

Primeiro é preciso expandir o símbolo inicial, usando uma das suas produções, depois pega-se numa subpalavra da palavra obtida e reescreve-se substituindo a subpalavra por uma palavra de acordo com as produções. O processo só acaba quando a palavra contiver apenas símbolos terminais.

Interpretando o Exemplo, identificam-se as produções:

$$P \longrightarrow \varepsilon | aPa | bPb$$

Eis uma derivação da palavra $aaabbb$:

$$P \Rightarrow aPb$$

$$aPb \Rightarrow aaPbb$$

$$aaPbb \Rightarrow aaaPbbb$$

$$aaaPbbb \Rightarrow aaa\varepsilon bbb$$

$$aaa\varepsilon bbb \Rightarrow aaabbb$$

2.3.2 Autómatos de pilha

Autómatos de Pilha são um tipo de autómato finito estendido que reconhece Linguagens Independentes de Contexto. Os autómatos de pilha possuem uma pilha que serve para armazenar símbolos, suportando operações de empilhamento e desempilhamento. Essa pilha fornece uma forma limitada de memória, permitindo que os autómatos de pilha reconheçam linguagens com estruturas mais complexas. Para executar corretamente a análise de uma palavra, é necessário ter memória (a pilha) para guardar partes da palavra que estão a ser reconhecidas [15].

Um autómato de Pilha é definido da seguinte forma:

$$A = \langle T, Q, Z, q1, z1, \delta, F \rangle$$

Onde:

- T — É o alfabeto de entrada
- Q — É o conjuntos de estados internos de A
- Z — É o alfabeto da pilha
- $q1$ — É o estado inicial de A, com $q1 \in Q$
- $z1$ — É o símbolo inicial da pilha, com $z1 \in Z$
- δ — É a relação de transição, $Q \times Z \times (T \cup \{\epsilon\}) \times Q \times Z^*$
- F — É o conjunto dos estados de aceitação, $F \subseteq Q$

Regra de transição

$$(q, z) \xrightarrow{a} (q', \epsilon)$$

Esta é a forma abreviada de $(q', \epsilon) \in \delta(q, z, a)$

Para haver uma transição é necessário a pilha não estar vazia, quando ocorre uma transição, desempilha-se o símbolo que está no topo e empilha-se uma palavra [13].

Operações sobre a pilha:

No caso não queremos desempilhar um símbolo na transição temos de o voltar a empilhar, desempilhamos z(símbolo no topo) e voltamos a empilhar z.

Se quisermos apenas empilhar uma palavra desempilhamos z e de seguida empilhamos λz, em que λ é uma palavra [13].

Apresentando um exemplo em que $A = \{a^n b^n | n \geq 1\}$:

$T = \{a, b\}$

$Q = \{p, q, t\}, q1 = p, F = t$

$Z = \{a, z\}, z1 = z$

Com as transições

$(p, z) \xrightarrow{a} (p, az)$,

$(p, a) \xrightarrow{a} (p, aa)$,

$(p, a) \xrightarrow{b} (q, \epsilon)$,

$(q, a) \xrightarrow{b} (q, \epsilon)$,

$(q, z) \xrightarrow{\epsilon} (t, z)$

Explicação da primeira transição: se o símbolo corrente do input for a, se o estado corrente for p e se z for o símbolo no topo da pilha, então é possível aplicar esta transição que nos conduz ao mesmo estado p com a adicionado ao topo da pilha e o símbolo a consumido no input.

Vamos ver se este autómato reconhece a palavras aaabbb:

Estado	Pilha	Fita
p	z	aaabbb
p	az	aabbb
p	aaz	abbb
p	aaaz	bbb
q	aaz	bb
q	az	b
q	z	λ
t	z	λ

Como foi atingido o estado final t e a fita está vazia significa que a palavra $aaabbb$ foi aceite.

A definição apresentada suporta autómatos não deterministas, sendo os autómatos deterministas um caso particular. Um aspeto muito curioso é que os autómatos de pilha não deterministas são mais poderosos que os deterministas. Concretamente, existem exemplos de linguagens que precisam mesmo de autómatos não deterministas para serem reconhecidas.

2.4 Linguagens dependentes de contexto

Linguagens dependentes de contexto são um tipo de linguagem formal utilizado em ciência da computação. Estas linguagens são definidas por gramáticas dependentes de contexto, nas quais as regras de produção são especificadas tendo em conta o contexto em que os símbolos aparecem. Esta capacidade de levar em consideração o contexto permite que as linguagens dependentes de contexto descrevam estruturas mais sofisticadas e representem informações mais complicadas [17].

2.4.1 Gramáticas dependentes de contexto

Ao contrário das gramáticas independentes de contexto, onde o lado esquerdo de uma produção corresponde apenas a um símbolo não terminal, uma gramática dependente de contexto pode ter do lado esquerdo uma sequência de símbolos não terminais e terminais. Uma gramática dependente de contexto é definido tal como as gramáticas livres de contexto pelo quádruplo $\langle V, T, P, S\rangle$. Com a diferença que P corresponde ao conjunto de produções, em que $P \subseteq V \times (T \cup V)^*$.

Nas definições mais comuns não permitido produções que encurtem uma palavra, ou seja se $\alpha \rightarrow \beta$ então $|\alpha| \leq |\beta|$.

Vendo agora o exemplo da gramática que gera a linguagem $\{a^n b^n c^n | n \geq 1\}$ O exemplo é o seguinte: [17]

$$\{S, A, B, C, W, Z\}, \{a, b, c\}, P, \{S\}$$

P vai ter as seguintes produções:
$S \rightarrow aBc$
$S \rightarrow aSBC$
$CB \rightarrow CZ$
$CZ \rightarrow WZ$
$WZ \rightarrow WC$
$WC \rightarrow BC$
$aB \rightarrow ab$
$bB \rightarrow bb$
$bc \rightarrow bc$
$cC \rightarrow cc$

Nesta gramática, podemos reparar que, para a variável B, existem três passos de derivações distintos consoante o contexto: a sequência "CB" é reescrita em "CZ" e a sequência "aB" é reescrita em "ab" e a sequência "bB" é reescrita "bb".

Para exemplificar, mostra-se a geração da palavra abc:

$$S \Rightarrow aBc$$

$$aBc \Rightarrow abc$$

2.4.2 Máquinas de Turing linearmente limitadas

Uma máquina de Turing linearmente limitada é um modelo computacional que reconhece as palavras que fazem parte de uma linguagem dependente de contexto, opera numa fita e oferece uma forma restrita de poder de computação. São semelhantes às máquinas de Turing, mas possuem uma quantidade limitada de espaço para armazenar informações. A fita de uma máquina de Turing linearmente limitada é dividida em casas, e a máquina pode ler e escrever símbolos na fita enquanto faz transições entre estados com base num conjunto de regras predefinidas. No entanto, as máquinas de Turing linearmente limitadas têm uma restrição crucial: a quantidade de espaço da fita usada durante a computação é limitada a uma função linear do tamanho da entrada [11].

Formalmente, podemos definir da seguinte forma: [13]

$$M = \langle T, Q, Z, q1, \delta, F \rangle$$

Onde

- Q — É o conjunto finito de estados
- Z — É o alfabeto da fita, contendo B(branco) e [,]
- T — É o alfabeto de entrada, com $T \subseteq Z$-{B}
- $q1$ — É o estado inicial, com $q_1 \in Q$

- F — É o conjunto e estados de aceitação
- δ — É a relação de transição (Q-F) $\times Z \times Q \times Z \times \{L, R\}$

Para além de B (branco) que é o símbolo que serve para preencher as casas que não foram preenchidas pela palavra de input, existe nas máquinas de Turing linearmente limitadas dois tipos de símbolo no alfabeto da fita, cuja função é delimitar a escrita na fita à esquerda ([ ) e à direita da fita. Quando é dada a instrução para a cabeça do controle se mover para além destes limites a cabeça simplesmente fica na mesma casa [11].

A definição de qualquer máquina de Turing linearmente limitada garante que não se ultrapassam os delimitadores nem se escreve sobre eles [11].

Apresentando agora um exemplo para a aceitação da palavra $aabbcc$ na linguagem $\{a^n b^n c^n | n \geq 1\}$.

O exemplo é o seguinte:

$$M = (\{a, b, c\}, \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b, c, x, y, z, B, [\,]\}, \{q_0\}, \delta, \{q_6\})$$

Com δ:

δ	a	b	c	x	y	Z	]
q_0	q_1, x, R				q_4, y, R		
q_1	q_1, a, R	q_2, y, R			q_1, y, R		
q_2		q_2, b, R	q_3, z, L			q_2, z, R	
q_3	q_3, a, L	q_3, b, L		q_0, x, R	q_3, y, L	q_3, z, L	
q_4					q_4, y, R	q_5, z, R	
q_5						q_5, z, R	$q_6,]\, R$

Na figura 2.3 apresenta-se a representação gráfica da máquina de Turing linearmente limitada que corresponde ao exemplo.

Dando o exemplo da palavra $aabbcc$ podemos testar se a máquina de Turing linearmente limitada a aceita, para este exemplo não sendo preciso mais casas do que as que correspondem à palavra vamos apenas limitar a fita ao número de casas da palavra.

$$[q_0 aabbcc] \vdash [x q_1 abbcc] \vdash [xa q_1 bbcc] \vdash [xay q_2 bcc] \vdash [xayb q_2 cc]$$

$$\vdash [xayb q_3 zc] \vdash [xay q_3 bzc] \vdash [xa q_3 ybzc] \vdash [x q_0 aybzc] \vdash [xx q_1 ybzc]$$

$$\vdash [xxy q_1 bzc] \vdash [xxyy q_2 zc] \vdash [xxyyz q_2 c] \vdash [xxyyzz q_3] \vdash [xxyyz q_3 z]$$

$$\vdash [xxyy q_3 zz] \vdash [xxy q_3 yzz] \vdash [xx q_0 yyzz] \vdash [xxy q_4 yzz] \vdash [xxyy q_4 zz]$$

$$\vdash [xxyyz q_5 z] \vdash [xxyyzz q_5] \vdash [xxyyzz] q_6$$

A máquina de Turing para num estado final e por isso a palavra é aceite.

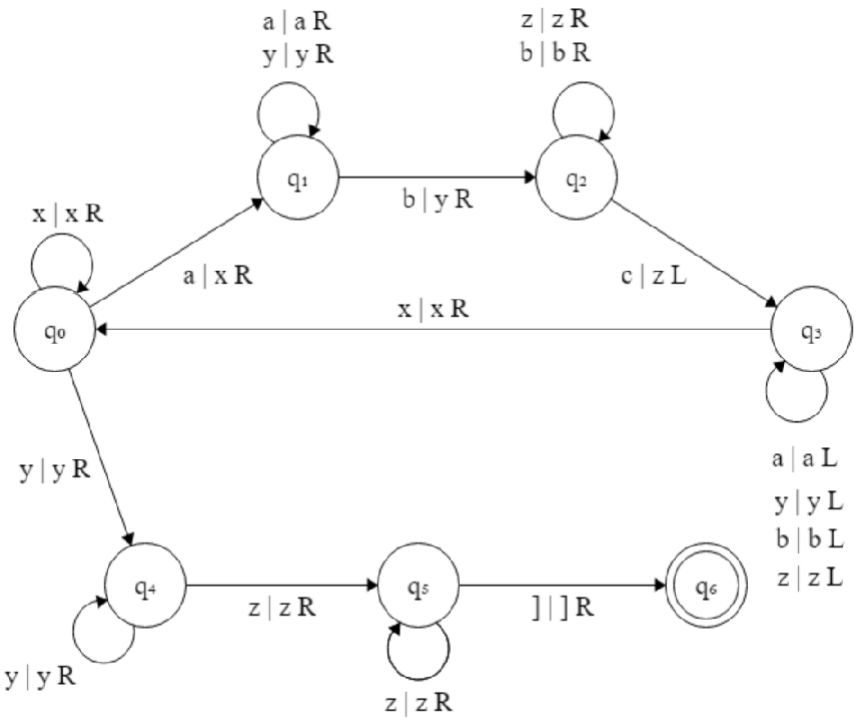

Figura 2.3: Máquina de Turing linearmente limitada

2.5 Linguagens recursivamente enumeráveis

As linguagens recursivamente enumeráveis são as linguagens de nível 0 e as últimas na hierarquia de Chomsky. Englobam todas as outras. São geradas por gramáticas irrestritas, e reconhecidas por máquinas de Turing. Se uma linguagem não for reconhecida por uma máquina de Turing não é recursivamente enumerável e também não é computável. As máquinas de Turing capturam a noção de computabilidade na sua forma mais geral. Apesar disso, uma máquina de Turing linearmente limitada está mais perto de um computador real visto ter uma memória limitada, enquanto que uma máquina de Turing tem memória infinita, o que não é possível no mundo real.

2.5.1 Gramáticas irrestritas

Uma gramática irrestrita é uma gramática que não apresenta restrições. É definida da mesma maneira que uma gramática dependente de contexto, $\langle V, T, P, S \rangle$, com a diferença que P corresponde ao conjunto de produções, em que $P \subseteq (V \cup T)^+ \times (V \cup T)^*$. [17]

Tal como as gramáticas dependentes de contexto, numa produção é possível ter qualquer tipo de símbolos(terminais e não terminais) tanto à esquerda como à direita. Ao contrário das gramáticas dependentes de contexto é possível haver produções que encurtem uma palavra, na verdade as gramáticas irrestritas apresentam apenas uma restrição, ε não pode aparecer à esquerda uma produção, todo o resto é permitido [11].

As gramáticas irrestritas são as gramáticas mais poderosas que geram as linguagens de tipo 0, ou seja todas as linguagens recursivamente enumeráveis [11].

As linguagens genuinamente de tipo 0 são complexas e estranhas e é difícil encontrar exemplos. Um exemplo de uma linguagem de tipo 0 que não é de tipo 1 é a linguagem do Problema da Correspondência de Post (PCP). Na referência [1] a linguagem do PCP é apresentada, acompanhada por uma linguagem irrestrita.

Eis outro exemplo de uma linguagem de tipo 0 que não é de tipo 1: o conjunto de todas as máquinas de Turing que reconhecem a palavra vazia. {M| M é uma máquina de Turing que reconhece a palavra vazia}. Note que o problema de testar se uma máquina de Turing pertence a esta linguagem é um problema semidecidível.

As linguagens de tipo 1 são todas decidíveis e são classificadas como recursivas. Curiosamente existem exemplos de linguagens decidíveis que não são de tipo 1. Portanto as fronteiras da decidibilidade não coincidem com as fronteiras da hierarquia de Chomsky.

2.5.2 Máquinas de Turing

A máquina de Turing é a forma mais poderosa de autómato, capaz de reconhecer palavras em linguagens de qualquer nível e executar qualquer processo computacional realizado por computadores modernos [11].

Uma máquina de Turing apresenta uma fita infinita dividida em casas, onde em cada casa cabe um símbolo, uma unidade de controlo finita e uma cabeça de leitura e escrita que aponta para uma casa [13].

O alfabeto da fita contém o símbolo B (branco), que é utilizado para preencher a fita. No início da computação, preenche-se a fita com a palavra que se deseja aceitar, colocando um símbolo em cada posição, e em seguida preenche-se todas as outras posições com B. Por convenção, a cabeça de escrita e leitura inicia-se na posição à esquerda do símbolo mais à esquerda da palavra.

Tal como as máquinas de Turing linearmente limitadas, definidas anteriormente, as máquinas de Turing são definidas da mesma forma:

$$M = \langle T, Q, Z, q1, \delta, F \rangle$$

Sendo que as máquinas de Turing não tem limite ao cumprimento da sua fita, não sendo necessário símbolos que delimitem o seu comprimento.

Olhando para o exemplo da Linguagem $\{w \in \{a,b\}^* : \#_a(w) = \#_b(w)\}$, vamos ter a máquina de Turing:

$$M = (\{a,b,c\}, \{q_0, q_1, q_2, q_3, q_4\}, \{a,b,c,x,B\}, \{q_0\}, \delta, \{q_4\})$$

com δ:

δ	a	b	x	B
q_0	q_1, x, R	q_3, x, R	q_0, x, R	q_4, B, R
q_1	q_1, a, R	q_2, x, L	q_1, x, R	
q_2	q_2, a, L	q_2, b, L	q_2, x, L	q_0, B, R
q_3	q_2, x, R	q_3, b, R	q_3, x, R	

Na figura 2.4 é possível ver a representação gráfica da máquina de Turing para o exemplo.

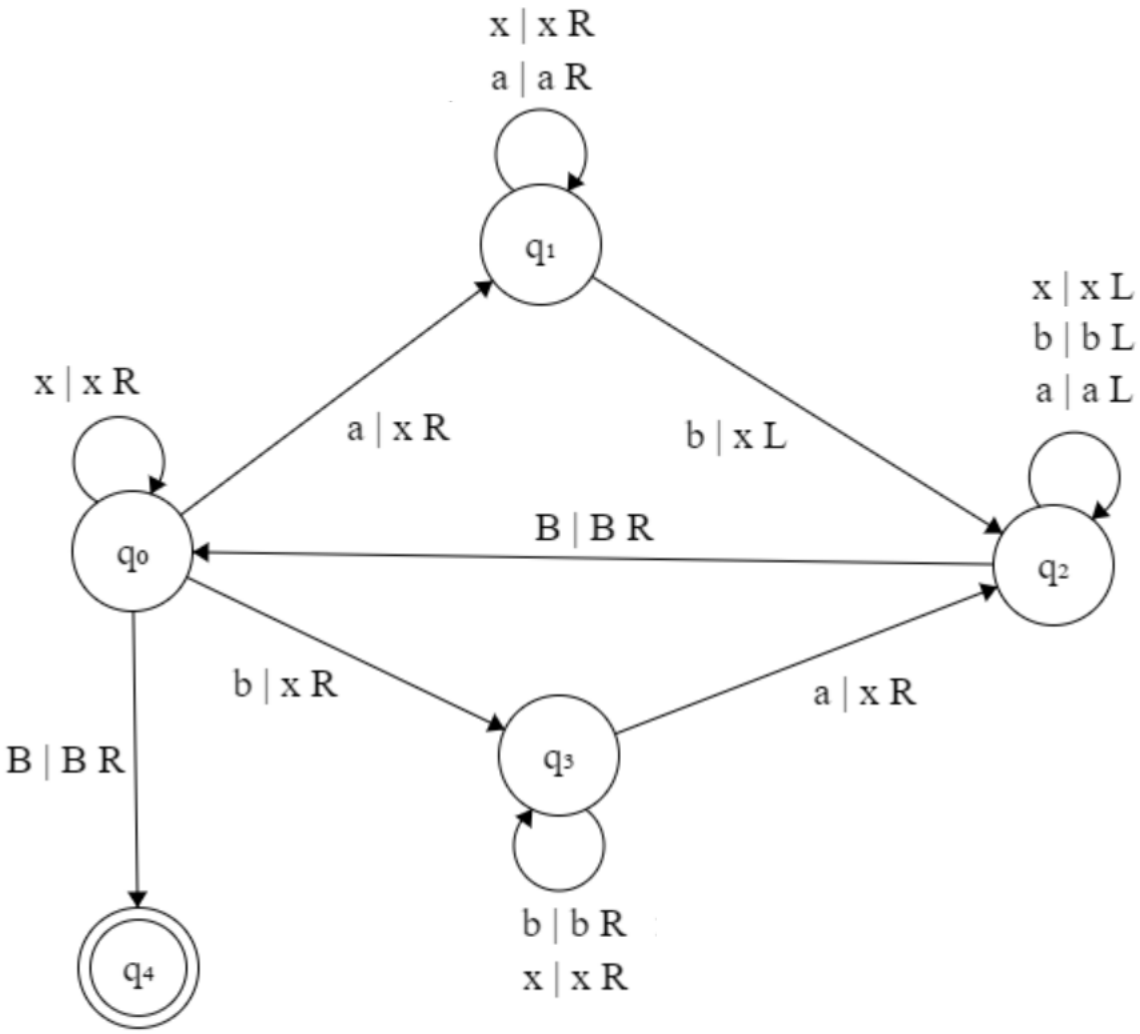

Figura 2.4: Maquina de Turing

Dando o exemplo da palavra $abba$ podemos testar se a máquina de Turing a aceita.

$$Bq_0abbaB \vdash Bxq_1bba \vdash Bxxq_2ba$$

$$\vdash Bxq_2xba \vdash Bq_2xxba \vdash Bq_0xxba$$

$$\vdash Bxq_0xba \vdash Bxxq_0ba \vdash Bxxxq_3a$$

$$\vdash Bxxxq_2x \vdash Bxxq_2xx \vdash Bxq_2xxx$$

$$\vdash Bq_2xxxx \vdash Bq_4xxxx$$

2.6 Composição de modelos

A composição de modelos é um tema importante que permite aprofundar o entendimento da teoria FLAT. É um tema abordado em muitos cursos de Teoria da Computação. As operações de composição que são normalmente estudadas em primeiro lugar são as chamadas operações regulares (união, concatenação e fecho de Kleene) mas podem ser acrescentadas outras operações como a complementação, interseção, etc.

Nesta área, duas questões importantes são: (1) que operações são fechadas, para cada tipo de modelo; (2) se é possível tomar um modelo composto e calcular um modelo atómico equivalente.

Para responder à primeira questão pode-se já referir que as três operações regulares são fechadas para todas as linguagens da hierarquia de Chomsky, as restantes operações como a complementação, a interseção, etc, já tem mais variabilidade quanto ao seu fecho nas diferentes linguagens. Para além destas três operações existem ainda outras operações que são fechadas para algumas linguagens mas não para todas [6].

Para as linguagens de nível 3, para além das operações regulares também a interseção, complementação, diferença, inverso e substituição de símbolos são fechadas. [17]

Para as linguagens de nível 2 para além das operações regulares, também são fechadas: a linguagem das palavras invertidas, a substituição de símbolos. Apesar de a interseção não ser fechada para as linguagens de nível 2, já é fechada a interseção entre uma linguagem independente de contexto e uma linguagem regular. O mesmo aplica-se à diferença [17].

Já para as linguagens de nível 1, são fechadas para a união, interseção, concatenação, substituição de símbolos, inverso e fecho de Kleene [21].

O conjunto das linguagens recursivamente enumeráveis ou nível 0, é apenas fechado para a união, interseção, concatenação e fecho de Kleene. Os outros casos já não são computáveis em geral, o que significa que já saem fora do nível 0. Por exemplo, a diferença entre duas linguagens recursivamente enumeráveis L_1 e L_2, tal que L_1-L_2 só é recursivamente enumerável se L_2 for recursiva e o complemento de L_1 é fechado, se e só se L_1 for recursivo [22].

2.7 Problemas de decisão

Um problema de decisão P é um problema computacional que se aplica a valores de entrada e cuja resposta é binária: sim ou não. A solução para o problema de decisão é um procedimento que procura determinar a resposta certa a cada pergunta do problema.

Em computação um problema de decisão é decidido por uma máquina de Turing, pois segundo a Tese de *Church-Turing* só existe um procedimento efetivo para resolver um problema de decisão P, se existir uma máquina de Turing que resolva P.

Interpretando o problema P como uma linguagem L_P, uma máquina de Turing M, **resolve** P, se para qualquer palavra M para e dá uma resposta, dizendo se a palavra pertence ou não a L_P. Neste caso L_P diz-se recursiva ou Turing-aceite e P diz-se decidível.

M **resolve parcialmente** P, se dada uma palavra que pertence a L_P, M para e dá a resposta "sim"; se dada uma palavra que não pertence a L_P, M pode parar com a resposta "não" ou então nunca parar. Neste caso L_P diz-se recursivamente enumerável ou Turing-reconhecível e P diz-se semidecidível [13].

O problema de paragem de uma máquina de Turing ("*Halting Problem*") refere o seguinte: existirá uma máquina de Turing M com capacidade para determinar a paragem de qualquer máquina de Turing durante o reconhecimento de uma qualquer palavra? A resposta é negativa e existe uma demonstração simples por contradição [17].

Um exemplo de uma linguagem que não é nem decidível, nem semidecidível é { M | M não reconhece a palavra vazia }, o que significa que esta linguagem não é computável.

3

Ferramentas de aprendizagem sobre teoria FLAT

Sendo a teoria de Linguagens Formais e Autómatos (FLAT - Formal Languages and Automata Theory) uma área frequentemente ensinada em cursos relacionados à informática, existe uma grande variedade de ferramentas que permitem estudar diferentes mecanismos ligados à teoria FLAT. No entanto, o tema da composição de modelos é escassamente abordado nessas ferramentas. Apenas algumas parecem oferecer recursos de composição de modelos, sendo que a maioria delas é voltada para autómatos finitos e geralmente não possuem uma representação gráfica. Portanto, podemos concluir que este é um tópico pouco explorado nas ferramentas de teoria FLAT.

3.1 JFLAP

O JFLAP (Java Formal Languages and Automata package) é um conjunto de ferramentas para experimentar com vários modelos da teoria da computação, incluindo autómatos finitos não deterministas, autómatos de pilha não deterministas, máquinas de Turing multifita, várias gramáticas, parsing e sistemas L. Para além de construir e testar exemplos para estes tópicos, o JFLAP permite também a conversão de um autómato finito não determinista para um autómato finito determinista, para um autómato finito mínimo, para uma expressão regular ou uma gramática regular [7]. JFAP é programado em Java, e vai atualmente na versão 7.1, sendo que a versão 8 ainda não se encontra acabada.

Começou a ser desenvolvido por volta de 1990 pelo Rensselaer Polytechnic Institute, sob a supervisão de Susan Rodger, e é a ferramenta mais conhecida e utilizada para teoria FLAT, apesar de das mais completas, esta não apresenta composição de modelos usando as operações regulares.

A aplicação abre logo com o menu principal 3.1 onde se pode escolher qual o modelo com que se pretende experimentar. A aplicação apresenta a opção "Turing Machine With Building Blocks", o que permite usar máquinas de Turing como macros noutras máquinas de Turing. É possível utilizar máquinas de Turing criadas pelo utilizador como macros ou então a documentação do JFLAP disponibiliza exemplos prontos a ser usados [8].

A seguir a clicar na opção "Turing Machine With Building Blocks" clicando no ícone do bloco à direita das setas é aberta uma janela para selecionarmos máquinas de Turing para usar como bloco 3.2.

Na figura 3.3 podemos ver a máquina de Turing usando os blocos disponibilizados pela documentação JFLAP.

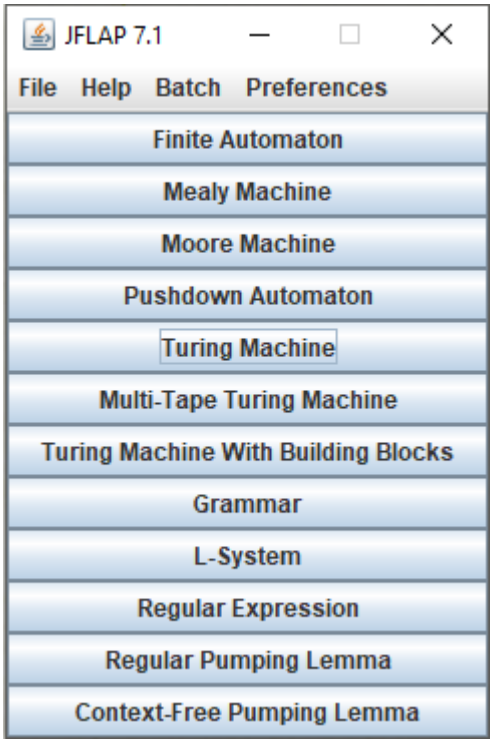

Figura 3.1: Menu inicial o JFLAP

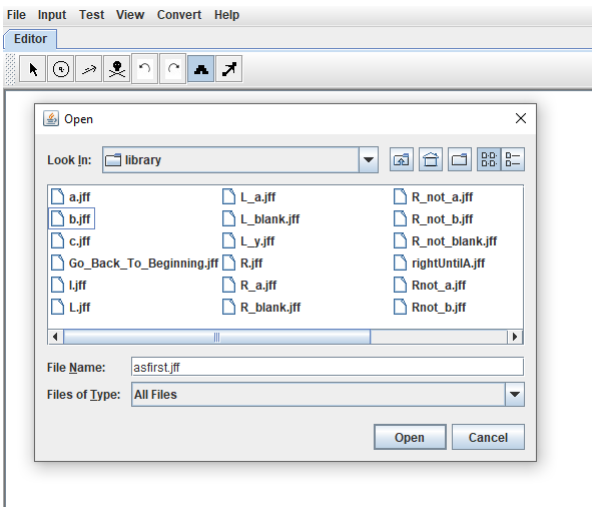

Figura 3.2: Escolher máquina de Turing a usar como bloco (JFLAP)

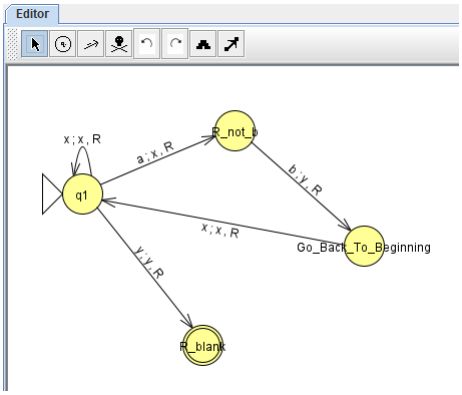

Figura 3.3: Máquina de Turing usando blocos (JFLAP)

3.2 FAdo

O sistema FAdo pretende fornecer uma biblioteca de software de código aberto, extensível e de alto desempenho para a manipulação simbólica de autómatos e outros modelos de computação. O principal objetivo é a pesquisa teórica e experimental, mas também a criação de uma ferramenta pedagógica para o ensino da teoria dos autómatos e linguagens formais.

Atualmente, o FAdo inclui a maioria das operações standard para a manipulação de linguagens regulares. O sistema implementa as expressões regulares e autómatos finitos deterministas e não deterministas como classes de Python. Para cada uma destas classes FAdo implementa as operações de união, interseção, concatenação, complementação, inverso e fecho de Kleene [3].

A interação com o sistema FAdo é feita através de programação em Python, não existindo uma interface gráfico, para poder ser utilizado é preciso importar os seguintes módulos:

- FAdo.fa - implementa a classe dos autómatos finitos
- FAdo.reex - implementa a classe das expressões regulares
- FAdo.fio -implementa métodos para IO de autómatos e modelos relacionados

Para além disso ainda apresenta um módulo (FAdo.comboperations) para operações combinadas como o concatenação com estrela($L_1.L_2*$) [2].

Para a composição de autómatos a ferramenta é muito simples de usar começa-se por definir em primeiro lugar os autómatos, indicando o alfabeto, os estados, o estado inicial e o estado final, e as transições. Os estados encontram-se numa lista, por isso sempre que é necessário referenciá-los(final, inicial, transições) usa-se o seu índex. Pode-se ver nas figuras 3.4, 3.5 os autómatos DFA's(Autómato finito determinista) m1 e m3.

Estando os dois autómatos criados é possível fazer operações sobre eles usando os métodos que se encontram na documentação do FAdo, na figura 3.6 podemos ver o

```
In [9]: >>> m1 = DFA()
   ...: >>> m1.setSigma(['a','b'])
   ...: >>> m1.addState('q0')
   ...: >>> m1.addState('q1')
   ...: >>> m1.addState('q2')
   ...: >>> m1.setInitial(0)
   ...: >>> m1.addFinal(2)
   ...: >>> m1.addTransition(0, 'a', 1)
   ...: >>> m1.addTransition(1, 'b', 1)
   ...: >>> m1.addTransition(1, 'a', 2)

In [10]: >>> m1.display()
```

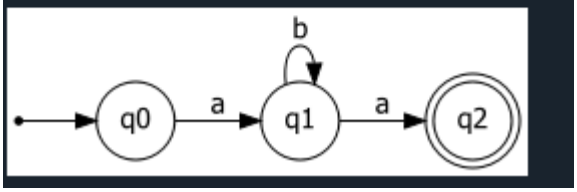

Figura 3.4: Autómato finito determinista m1 no FAdo

```
In [7]: >>> m3 = DFA()
   ...: >>> m3.setSigma(['a','b'])
   ...: >>> m3.addState('q0')
   ...: >>> m3.addState('q1')
   ...: >>> m3.addState('q2')
   ...: >>> m3.setInitial(0)
   ...: >>> m3.addFinal(2)
   ...: >>> m3.addTransition(0, 'a', 1)
   ...: >>> m3.addTransition(1, 'a', 1)
   ...: >>> m3.addTransition(1, 'b', 2)
   ...: >>> m3.addTransition(2, 'b', 2)

In [8]: >>> m3.display()
```

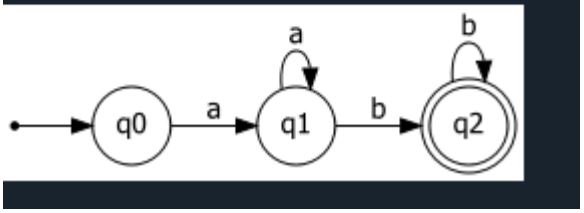

Figura 3.5: Autómato finito determinista m3 no FAdo

resultado da concatenação de m1 a m3, usamos a função *concatI* que realiza a concatenação de dois DFA's, existe também a função *concat* que realiza a concatenação de dois DFA's e se algum deles não for completo, ele é completado [14]. Em ambas as funções é preciso definir o valor booleano *strict* que especifica se os alfabetos vão ser verificados, se metermos a *true*, os alfabetos são verificados e se não forem iguais é lançada uma exceção, caso contrário não são verificados.

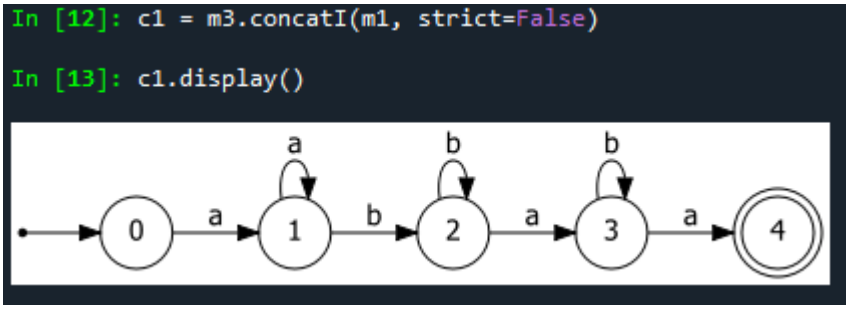

Figura 3.6: Concatenação dos autómatos finitos deterministas m3 e m1 no FAdo

Com este exemplo é possível que o FAdo é uma plataforma que apesar de apenas suportar as operações entre linguagens regulares é um sistema para retirar ideais para esta tese, sendo uma das poucas plataformas que suportam operações sobre modelos FLAT.

3.3 FAT

FAT(*Finite Automata tool*) é uma ferramenta que implementa vários algoritmos sobre autómatos finitos deterministas e não deterministas, como a determinação e a minimização. A ferramenta também inclui as operações de interseção, união e complemento. Apesar de ter poucas operações e serem só sobre autómatos finitos, o FAT tem a uma interface gráfica desenvolvida e tem a particularidade de resolver as operações passo a passo.

Para criar um autómato, é preciso ir a *File* e clicar em *New automaton* para abrir a janela onde se vai especificar um novo autómato(figura 3.7), nesta ferramenta os autómatos DFA é preciso haver transições para todos os símbolos do alfabeto a sair de todos os estados. [4]

Os autómatos criados ficam guardados na lista do lado direito do ecrã, ao selecionar dois autómatos é possível selecionar a interseção ou união(figura 3.8), ao selecionar um DFA é possível escolher o complemento.

De seguida a interseção ou união são realizadas passo a passo, sendo possível escolher a velocidade e ficando registados os passos que foram feitos.(figura 3.9)

Apesar da poucas operações que suporta o FAT tem uma parte gráfica bastante desenvolvida, o que será útil de ter em conta para a parte gráfica desta tese.

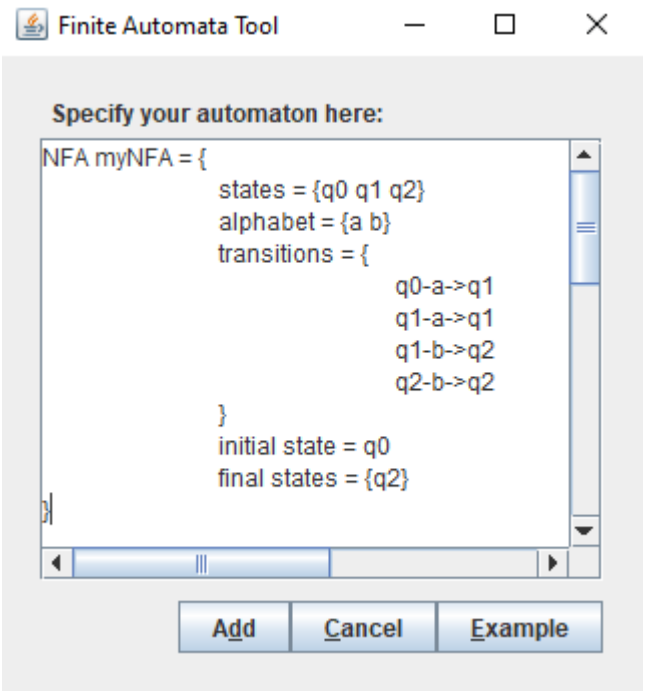

Figura 3.7: Criação de um autómato na plataforma FAT

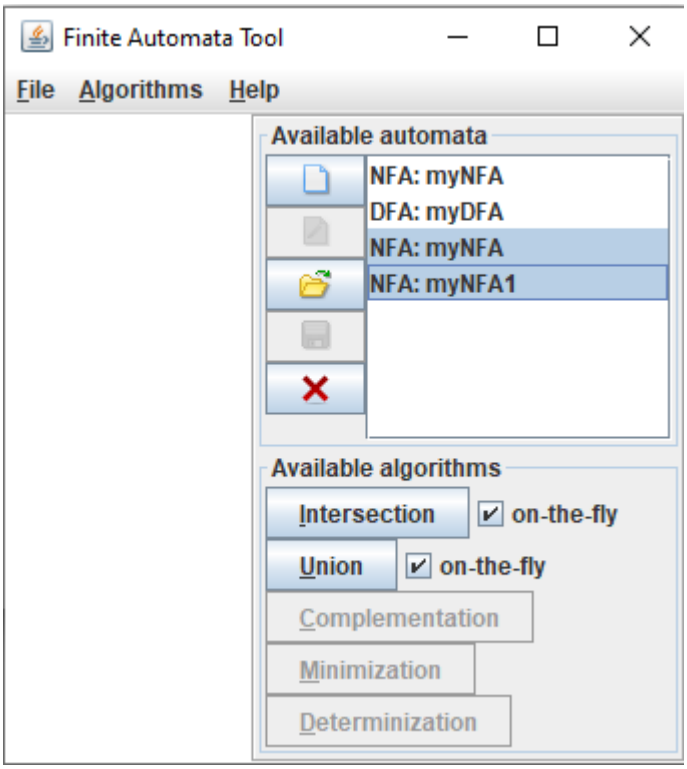

Figura 3.8: Selecionar dois autómatos no FAT

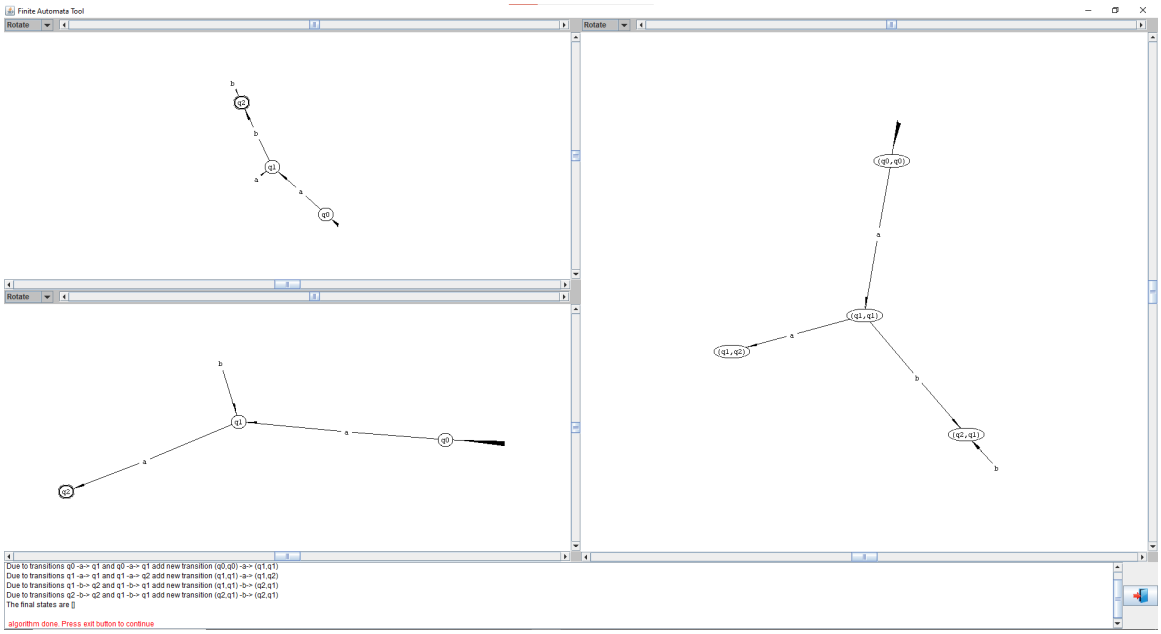

Figura 3.9: Interseção de dois autómatos no FAT

4

Plataforma OFLAT e OCaml-FLAT

A biblioteca OCamlFLAT e a plataforma OFLAT são os principais componentes desta tese, ambas desenvolvidas em OCaml. Encontram-se já num estado avançado de desenvolvimento fruto do trabalho de vários alunos e professores que passaram por este projeto. A biblioteca OCamlFLAT é composta por vários módulos que compõem a parte lógica do sistema, onde já estão implementados vários modelos de teoria FLAT. A plataforma OFLAT é uma aplicação web que apresenta a interface gráfica do sistema.

4.1 OCamlFLAT

A biblioteca OCamlFLAT encontra-se organizada por módulos para ficar mais organizada e estruturada. Existem módulos que implementam modelos da teoria FLAT como expressões regulares, autómatos finitos, gramáticas independentes de contexto, e módulos auxiliares, como testes, erros, parsers. [5] De notar que os módulos seguem uma relação hierárquica entre eles.

Alguns dos módulos mais importantes são [19, 18]:

Entity - é o módulo que representa o nível mais abstrato da biblioteca, as entidades são os exercícios e os modelos FLAT, que contém o tipo, uma descrição e um ID.

Exercise - é o módulo que suporta exercícios, validados com UnitTests. Estende Entity.

Model - é um módulo abstrato que estende Entity e introduz funcionalidades comuns aos modelos FLAT disponíveis na biblioteca. Exemplos de métodos são o $validate$ e o $accept$.

Exemplos de módulos que representam modelos FLAT e estendem $Model$ [16]:

RegularExpression - é o módulo que contém as funcionalidades de uma expressão regular

FiniteAutomaton - é o módulo que contém as funcionalidades de um autómato finito

ContextFreeGrammar - é o módulo que contém as funcionalidades de uma gramática livre de contexto

PushdownAutomaton - é o módulo que contém as funcionalidades de um autómato de pilha

4.2 O tratamento do infinito potencial na biblioteca OCamlFLAT

A teoria FLAT, com os seus diversos reconhecedores e geradores de linguagens formais constitui um ramo da Matemática. Não se deve estranhar o facto da noção de infinito, ou talvez mais rigorosamente infinito potencial, ocorrer várias vezes nas definições. Alguns exemplos: nos autómatos finitos e nos autómatos de pilha, podem ocorrer loops no grafo das transições; nas gramáticas independentes de contexto permite-se o uso recursividade; nas máquinas de Turing, a fita é infinita e também podem ocorrer loops no grafo das transições; para a generalidade dos modelos, quando se procura reconhecer ou gerar uma palavra usando diretamente as definições da teoria, ficamos automaticamente com uma árvore infinita de configurações para explorar.

No entanto, surgem problemas quando queremos programar num computador operações onde o infinito potencial intervém. Consideremos o problema de explorar sistematicamente, usando procura em largura, uma árvore infinita de configurações. Podemos garantir terminação no caso em que a resposta teórica seja positiva; mas a função pode não terminar no caso da resposta teórica ser negativa - e se não termina, ficamos sem saber qual é a resposta.

Note que a dificuldade descrita decorrem de transição do domínio da Matemática para o domínio da Computação teórica. Mas ainda resta o problema da passagem do domínio da Computação teórica para o domínio da Computação prática. Mesmo nos casos em que a função de reconhecimento teoricamente termina, pode não existir suficiente memória no computador para se atingir a terminação, ou podemos não querer esperar mais por uma resposta que está demasiado demorada.

No OCamlFLAT, a função genérica de reconhecimento de palavras termina sempre. Pode terminar com um resultado positivo. Pode terminar com um resultado negativo. Mas também pode terminar com um resultado indefinido, o que representa alguma de três situações: (1) o resultado não é computável (seria teoricamente negativo, mas demoraria tempo infinito a determinar isso); (2) a memória reservada para a execução esgotou-se; (3) o tempo reservado para a execução esgotou-se.

Felizmente, para alguns dos modelos há formas de programar a função de reconhecimento duma forma que garante terminação teórica. Por exemplo, no caso dum autómato finito, podemos converter o autómato para a forma determinista (portanto, sem transições vazias) e depois usar o algoritmo de reconhecimento geral. Mas, no caso das máquinas de Turing não há nada a fazer porque se trata dum mecanismo muito poderoso e a semi-decidibilidade do procedimento de reconhecimento é intrínseca.

4.3 Plataforma OFLAT

A plataforma OFLAT corresponde à parte gráfica do projeto, estando online disponível ao público. Tem como principal objetivo servir uma ferramenta pedagógica relativamente simples de usar. Apresenta uma barra lateral à esquerda com um menu estático e um

espaço de trabalho em branco dinâmico à direita, cuja vista muda conforme as opções selecionadas no ecrã. Na figura 4.1 é possível ver o ecrã principal quando clicamos na opção *Novo modelo -> Autómatos finitos* na barra lateral

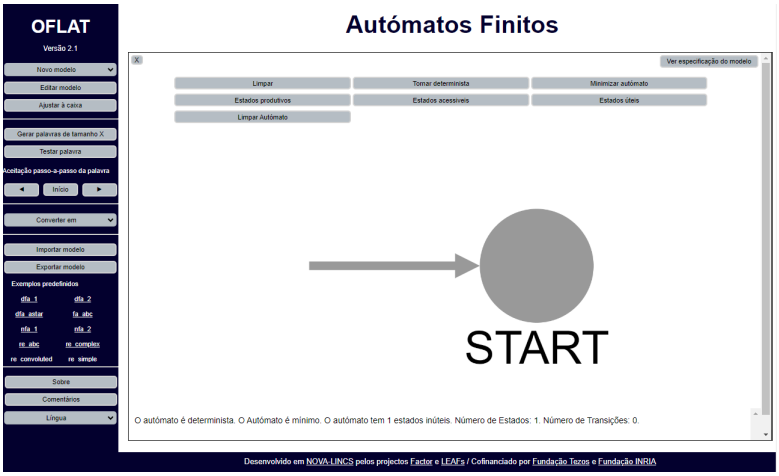

Figura 4.1: Ecrã inicial do OFLAT para criação de novos autómatos finitos

A plataforma é fácil de usar e intuitiva, por exemplo, para criar um autómato finito é preciso primeiro escolher a opção na barra lateral de seguida clicar com o botão direito do rato no espaço em brando onde aparece a opção para criar um estado, podendo ser final ou inicial(figura 4.2) sendo que já aparece um estado inicial no ecrã. Ao carregar com o botão direito do rato o estado é possível adicionar transições(figura 4.3) e assim construir um autómato finito no canto inferior esquerdo do ecrã branco aparece informação sobre o autómato, se é determinista, se é mínimo , o número de estados inúteis que têm, o número de estados e transições com figura 4.4 podemos ver que o autómato criado é determinista e não mínimo, podemos assim clicar na opção *minimizar autómato* para nos ser mostrada uma versão mínima. 4.5

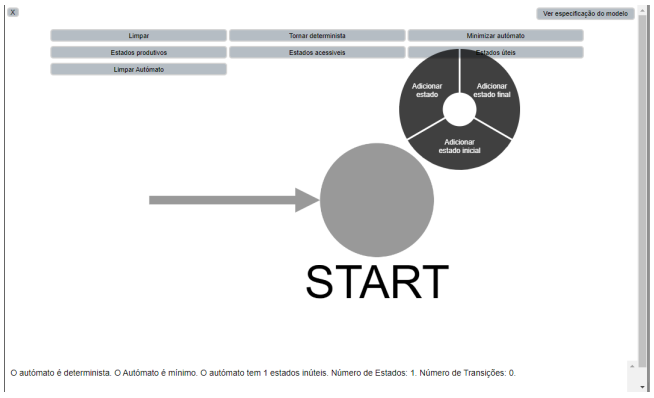

Figura 4.2: Criação de um novo estado de um autómato finito no OFLAT

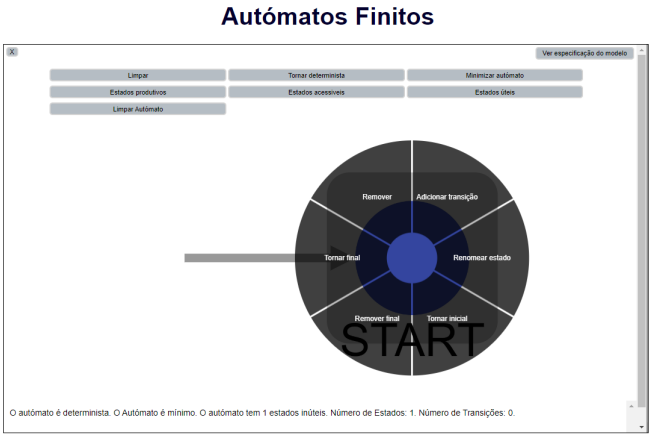

Figura 4.3: Criação de um a nova transição de um autómato finito no OFLAT

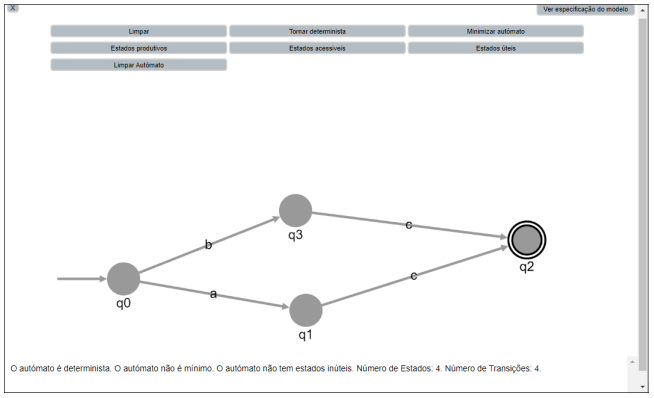

Figura 4.4: Autómato finito criado no OFLAT

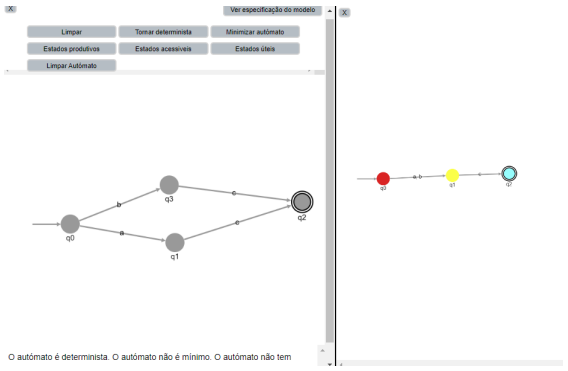

Figura 4.5: Minimização autómato finito criado no OFLAT

5

Tecnologia

5.1 Interoperabilidade OCaml/JavaScript

A interoperabilidade entre OCaml e JavaScript é possível através da utilização da ferramenta js_of_ocaml que traduz código de OCaml para JavaScript, que permite correr código OCaml num browser. Apresenta três componentes: [19]

- js_of_ocaml-ppx: Um pré-processador que extende a biblioteca OCaml
- js_of_ocaml library: uma biblioteca que contêm bindings de diversas categorias de objetos JavaScript
- js_of_ocaml compiler: traduz código com formato OCaml para JavaScript

5.1.1 Tipos JavaScript

Os tipos em OCaml e JavaScript não se representam da mesma maneira em OCaml e JavaScript, por exemplo, uma string em OCaml é uma array mutável de bytes, enquanto que em JavaScript são um array constante de UTF-16 pontos de código. Por isso deve existir uma maneira de representar os tipos de JavaScript em OCaml. Na tabela pode-se ver as correspondências entre as linguagens. [9]

OCaml	JavaScript em OCaml	JavaScript
int	int	Number
float	float	Number
bool	bool Js.t	Boolean
string	Js.js_string Js.t	String
array	Js.js_array Js.t	Array

5.1.2 Bindings

A existência e criação de bindings no js_of_ocaml permite usar código e bibliotecas JavaScript em OCaml de maneira tipificada. Apesar de o js_of_ocaml já ter muitos bindings

úteis, é ainda possível criar novos bindings que permitem a interação com outras bibliotecas e objetos JavaScript que sejam relevantes para o projeto. Um exemplo de um binding bastante útil é o de uma tabela como se pode ver no exemplo a seguir:

Listagem 5.1: Excerto do binding da tabela(*tableElement*).

```
1        class type tableElement = object
2        inherit element
3
4        method caption : tableCaptionElement t prop
5        method tHead : tableSectionElement t prop
6        method tFoot : tableSectionElement t prop
7        method rows : tableRowElement collection t readonly_prop
8        method tBodies : tableSectionElement collection t readonly_prop
9        method align : js_string t prop
10       method createCaption : tableCaptionElement t meth
11       method deleteCaption : unit meth
12       method insertRow : int -> tableRowElement t meth
13       method deleteRow : int -> unit meth
14     end
15
```

5.1.3 Módulos js_of_ocaml

O js_of_Ocaml possui já vários módulos com bindings pré-definidos, no entanto, só alguns é que são necessários para o desenvolvimento do OFLAT. [18] Alguns exemplos desses módulos e submódulos são:

- Módulo Js - contém as definições dos tipos JavaScript, funções sobre objetos standard JavaScript e funções de conversão entre tipos OCaml e JavaScript
- Submódulo Js.Unsafe - permite operações sobre JavaScript que não são seguras
- Módulo Dom_html - Define elementos do DOM HTML
- Módulo Firebug - permite visualizar objetos na consola em modo programador no browser.

5.2 Cytoscape

Cytoscape é uma plataforma de software open source que apesar de inicialmente ter sido desenhada para pesquisa biológica, hoje em dia é uma plataforma geral para visualização e análise de grafos. Neste projeto é usada a plataforma OFLAT para a visualização dos vários modelos suportados pela aplicação. [20] Para usar esta biblioteca é necessário criar os bindings necessários para interoperabilidade OCaml/JavaScript.

6

Suporte para composição de modelos na biblioteca OCamlFLAT

O objetivo desta dissertação, ao nível da biblioteca, consistiu em adicionar suporte para composição de modelos usando as operações regulares e a operação de interseção. Foi assim adicionado o módulo *Composition*, que introduz uma representação para composições (modelos compostos), assim como numerosas funções de manipulação dessa representação. A funcionalidade mais ambiciosa do módulo consiste numa operação de cálculo dum modelo atómico com o tipo desejado a partir dum qualquer modelo composto — por exemplo, o cálculo dum autómato de pilha a partir da união entre um autómato finito e o fecho de Kleene duma gramática independente de contexto.

Foi também criado um módulo chamado *Repository* que armazena modelos, associando um nome a cada um desses modelos. Numa composição é conveniente identificar, com nomes, os modelos constituintes, sendo isso especialmente importante na aplicação gráfica.

6.1 Conceitos básicos

A implementação da composição de modelos na biblioteca, suporta os cinco modelos clássicos já desenvolvidos em dissertações anteriores: expressões Regulares(RE), autómatos finitos(FA), autómatos de pilha(PDA), gramáticas livres de contexto(CFG) e máquinas de Turing(TM).

É útil introduzir alguma nomenclatura associada ao novo conceito de composição. Chamamos "modelo atómico" a qualquer um dos cinco modelos primitivos atrás referidos; "modelo composto" ou "composição" a qualquer modelo que resulte da composição de vários modelos: atómicos ou compostos.

Chamamos "composição homogéneas" a uma composição em que todos os modelos atómicos constituintes são do mesmo tipo; "composição mista" a uma composição em que ocorrem pelo menos dois modelos atómicos com tipos diferentes.

Uma composição (modelo composto) é representada por uma árvore estrutural (Listagem 6.1) sobre a qual podem ser definidas diversas funções úteis. Algumas dessas funções,

por exemplo o *accept*, podem ser implementadas diretamente, sem ser necessário efetuar o pesado cálculo prévio dum modelo atómico equivalente.

Listagem 6.1: Definição de composição no OCamlFLAT.

```
1     type t =
2     | Plus of t * t
3     | Seq of t * t
4     | Intersect of t * t
5     | Star of t
6     | FA of FiniteAutomaton.t
7     | RE of RegularExpression.t
8     | CFG of ContextFreeGrammarBasic.t
9     | PDA of PushdownAutomaton.t
10    | TM of TuringMachine.t
11    | TMM of TurMachMultiTypes.t
12    | FAO of FiniteAutomaton.model
13    | REO of RegularExpression.model
14    | CFGO of ContextFreeGrammarBasic.model
15    | PDAO of PushdownAutomaton.model
16    | TMO of TuringMachine.model
17    | Rep of string
18    | Comp of t
19    | Empty
20    | Zero
21
```

Quando se discutem as operações de composição de modelos, agora no contexto de cálculo de novos modelos atómicos, convém começar por considerar apenas a situação mais simples que envolve apenas modelos do mesmo tipo. Concretamente, no caso das operações binárias, por exemplo, a união, considera-se que os dois elementos participantes são do mesmo tipo. Numa operação unária, como o fecho de Kleene, o problema não se coloca. Neste contexto também é importante discutir a questão de saber se uma operação de composição é ou não fechada para um dado tipo de modelo.

Numa segunda fase já se pode discutir o problema de aplicar uma operação de composição binária a dois modelos de tipo diferente. Neste caso a solução implica simplesmente converter os modelos para o mesmo tipo antes de efetuar a composição. Portanto, o uso de conversões de tipo desempenha um papel essencial no contexto da avaliação duma composição mista.

Neste trabalho implementam-se todas as operações de composição com argumentos homogéneos; também se usam conversões de tipo (a maioria das quais já estava disponível na versão anterior da biblioteca), e existe uma operação geral de cálculo dum modelo composto que usa com habilidade os elementos que foram referidos.

6.2 Implementação na biblioteca

As operações regulares: união, concatenação e fecho de Kleene são suportadas por todos os modelos, pois são fechadas para todos. As operações de união e concatenação são operações binárias que recebem dois modelos do mesmo tipo *a* e *b* e devolvem um modelo de tipo igual. Já o fecho de Kleene é uma operação unária que recebe só um modelo *t* e devolve um modelo do mesmo tipo. Para estas operações foi tido como base os slides do professor Luís Monteiro onde está explicado como fazer a composição das operações regulares em autómatos finitos e a partir daí foi possível aplicar a mesma lógica para os restantes modelos [13].

Foi ainda implementada a operação de interseção nos autómatos finitos, expressões regulares e máquinas de Turing. Como para os autómatos de pilha e gramáticas livres de contexto a interseção não é fechada, não foi aplicada a esses modelos. Tal como a união e concatenação também a interseção é uma função binária.

As funções mais importantes que podem ser usadas externamente à biblioteca são o *accept*, o *generate* e o grupo de funções *eval*. Cada tipo de modelo tem a sua própria função *eval*.

O modelo atómico resultante da composição depende da função *eval* chamada. Para a função *evalFA* a composição resultante vai ser um autómato finito, já se a função chamada for a *evalCFG* a composição vai dar origem a uma gramática livre de contexto.

Foram também usados vários módulos de apoio que já existiam na biblioteca como o módulo *Set*, que contém muitas operações importantes de manipulação de sets, e o módulo *PolyBasic* que suporta a conversão de modelos.

O tipo de dados da composição é definido no *CompositionSupport* e permite suportar as operações de composição e as diferentes representações dos modelos, incluindo representado pelo seu nome.

6.3 Accept e Generate

O *accept* e o *generate* são funções fundamentais na biblioteca e estão presentes em todos os modelos. São funções recursivas que recebem uma composição de modelos. O *accept* recebe uma composição e uma palavra e vai verificar se a palavra é aceite ou não pela composição, retornando verdadeiro ou falso. O *generate* recebe uma composição e um número inteiro, e gera todas as palavras da linguagem da composição que tenham esse comprimento.

Têm ambas a mesma estrutura recursiva que tem por base os diferentes modelos possíveis na composição. Quando se trata de modelos únicos são chamados o *accept* ou *generate* dos próprios modelos que já existem na biblioteca.

Para as operações de composição o *accept* e o *generate* são chamados de maneira diferente.

6.3.1 Accept

O *Accept* processa uma composição e faz uma análise para cada operação.

Para uma palavra ser aceite por uma união é necessário que ela seja aceite por pelo menos um dos modelos que compõem a união.

Para uma palavra ser aceite por uma interseção é necessário que ela seja aceite por todos os modelos que pertencem à interseção.

Para uma palavra ser aceite pela concatenação é necessário que a primeira parte da palavra seja aceite pelo primeiro modelo da concatenação e a segunda parte seja aceite pelo segundo modelo. É necessário testar todas as possibilidades de partição da palavra, e se houver pelo menos uma que corresponda às afirmações anteriores a palavra é aceite.

Para uma palavra ser aceite pelo fecho de Kleene(Star t) é necessário que haja pelo menos uma maneira de partir a palavra em uma ou mais partes e que todas as partes sejam aceites pelo t, ou então a palavra ser vazia.

Listagem 6.2: Excerto de código de *accept*.

```
1       let rec accept (c: t) (w: word) : bool =
2       match c with
3       | Plus (a,b) ->
4         accept a w || accept b w
5       | Intersect (a,b) ->
6         accept a w && accept b w
7       | Seq (a,b) ->
8         let list = Set.make(decompositions w) in
9         Set.exists (fun (p0,p1) -> (accept a p0) && (accept b p1)) list
10      | Star t ->
11        w = []
12        || (let list = Set.remove ([],w) (Set.make(decompositions w)) in
13            Set.exists (fun (p0,p1) -> (accept t p0) && (accept (Star t) p1))
    list)
14
```

6.3.2 Generate

O *Generate* vai processar uma composição e fazer uma análise para cada operação.

Na união, o resultado é a união dos conjuntos de palavras geradas pelas linguagens dos modelos pertencentes à composição

Na interseção, o resultado é a interseção dos conjuntos de palavras geradas pelas linguagens dos modelos pertencentes à composição.

Na concatenação, o resultado é o conjunto de todas as palavras cuja primeira parte foi gerada pelo primeiro modelo da concatenação e a segunda parte foi gerada pelo segundo modelo da composição, sendo que as duas partes juntas não devem exceder o tamanho máximo dado.

No fecho de Kleene (Star t), são geradas todas as palavras de t. Ao conjunto dessas palavras é aplicada a operação de fecho de Kleene da teoria de conjuntos.

Listagem 6.3: Excerto de código de *generate*.

```
1     let rec generate (c: t) (len: int) : words =
2     match c with
3     | Plus (a,b) ->
4       Set.union (generate a len)  (generate b len)
5     | Intersect (a,b) ->
6       Set.inter (generate a len) (generate b len)
7     | Seq (a,b) ->
8       let left = generate a len in
9       let rigth w = generate b (len - (List.length w)) in
10      let conc w = concatAllS w (rigth w) in
11        Set.flatMap (fun lw -> conc lw) left
12    | Star t ->
13      let exp = generate t len in
14        Set.star exp len
15
```

6.4 Autómatos finitos

6.4.1 União

A união de dois autómatos finitos é uma operação relativamente simples. Vai existir um estado novo "ligado" a cada um dos autómatos, sendo assim possível percorrer qualquer um dos dois autómatos (figura 6.1).

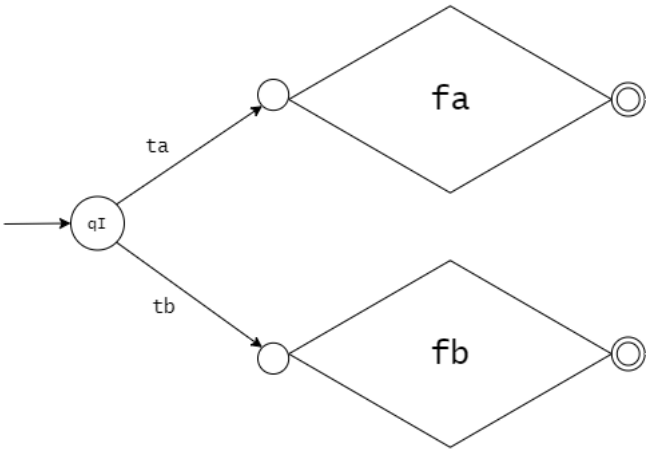

Figura 6.1: Representação da união de dois autómatas finitos

Em primeiro lugar cria-se um estado novo. Este estado novo vai ser o estado inicial do novo autómato gerado. São criadas duas transições novas vazias, que saem do novo estado inicial e vão cada uma para o estado inicial de cada autómato. Assim a transição 'ta' vai ligar ao estado inicial ao autómato 'fa' e a transição 'tb' vai ligar o estado inicial ao autómato 'fb'.

O alfabeto deste novo autómato vai ser a união dos alfabetos dos dois autómatos, tal como os estados de aceitação iram ser a união dos estados de aceitação. O conjunto de estados vai ser a união do conjunto de estados de cada autómato com a adição do estado criado, enquanto este passa a ser o estado inicial do novo autómato.

Já o conjunto de transições do novo autómato passa a ser a união dos conjuntos de transições dos dois autómatos adicionando as duas transições novas criadas.

Listagem 6.4: Função de união de autómatos finitos *evalPlusFA*.

```
1  let evalPlusFA (fa: FiniteAutomaton.t) (fb: FiniteAutomaton.t):
     FiniteAutomaton.t =
2
3    let qI = str2state (IdGenerator.gen("q")) in
4    let ta = (qI, epsilon,fa.initialState) in
5    let tb = (qI,epsilon,fb.initialState) in
6    {
7      alphabet = Set.union fa.alphabet fb.alphabet;
8      states = Set.add qI (Set.union fa.states fb.states);
9      initialState = qI;
10     transitions = Set.add tb (Set.add ta (Set.union fa.transitions fb.
     transitions));
11     acceptStates = Set.union fa.acceptStates fb.acceptStates
12   }
13
```

6.4.2 Concatenação

A concatenação de dois autómatos finitos é a sequência do segundo a seguir ao primeiro, ou de 'fa' seguido de 'fb' (figura 6.2).

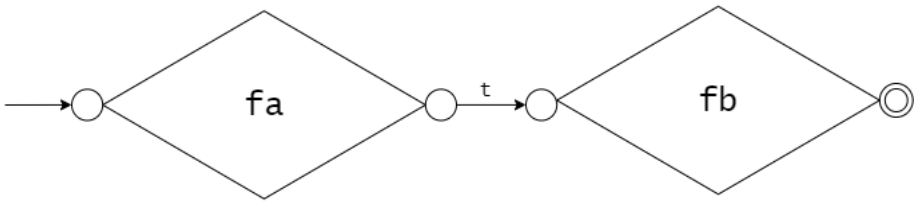

Figura 6.2: Representação da concatenação de dois autómatas finitos

Para esta operação foi criada a função auxiliar *createTransitionFA* que cria uma transição vazia entre dois estados. Esta função é necessária, pois é preciso "ligar" os estados de aceitação do primeiro autómato com o estado inicial do segundo autómato. Assim é criado um conjunto de transições usando um *map* que percorre todos os estados de aceitação de 'fb' e chama a função auxiliar que "liga" cada estado de aceitação ao estado inicial de 'fa'. O conjunto de transições do novo autómato passa assim a ser a união das transições dos dois autómatos com a adição do conjunto de transições criado.

O alfabeto do novo autómato tal como na operação de união vai ser a união dos alfabetos dos dois conjuntos, também o conjunto de estados passa a ser a união dos conjuntos de

estados dos dois autómatos. O estado inicial vai ser o estado inicial do primeiro autómato, 'fa' e os estados de aceitação vão ser os estados de aceitação do segundo autómato, 'fb'.

Listagem 6.5: Função de concatenação de autómatos finitos *evalSeqFA*.

```
1  let evalSeqFA (fa: FiniteAutomaton.t) (fb: FiniteAutomaton.t):
     FiniteAutomaton.t =
2
3    let t = Set.map (fun st -> createTransitionFA st fb.initialState) fa.
     acceptStates in
4    {
5      alphabet = Set.union fa.alphabet fb.alphabet;
6      states = Set.union fa.states fb.states;
7      initialState = fa.initialState;
8      transitions = Set.union t (Set.union fa.transitions fb.transitions);
9      acceptStates = fb.acceptStates
10   }
11
```

6.4.3 Fecho de Kleene

O fecho de Kleene é a repetição do autómato zero ou mais vezes (figura 6.3).

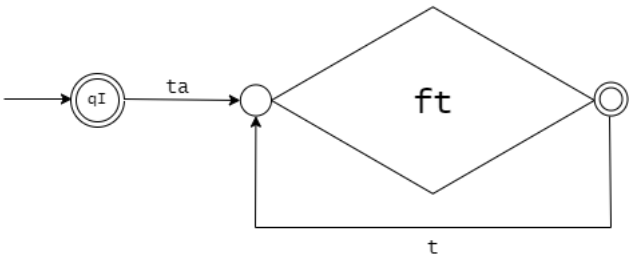

Figura 6.3: Representação do fecho de Kleene de um autómata finito

O fecho de Kleene nos autómatos finitos funciona de maneira semelhante à concatenação. É usada a mesma função auxiliar *createTransitionFA*, que cria transições vazias entre dois estados para ligar os estados de aceitação do único autómato, que na função chamamos 'ft', que recebe e "liga" ao estado inicial do mesmo, criando um conjunto de transições vazias.

Como o fecho de Kleene exige que a transição vazia seja aceite é necessário ainda criar um estado novo e uma transição vazia,'t', que vai deste estado para o estado inicial de 'ft' e o estado novo passa a pertencer ao conjunto de estados de aceitação.

O alfabeto vai ser igual ao alfabeto de 'ft', o conjunto de estados vai o mesmo que de 'ft' com a adição do estado novo, que também vai ser o estado inicial. O conjunto de transições vai ser a união das transições de 'ft' com o conjunto de transições vazias com a adição da transição vazia 't'. O conjunto de estados de aceitação é igual ao de 'ft' adicionando o estado inicial novo.

Listagem 6.6: Função de fecho de Kleene de autómatos finitos *evalStarFA*.

```
1  let evalStarFA (ft: FiniteAutomaton.t) : FiniteAutomaton.t =
2
3    let qI = str2state (IdGenerator.gen("q")) in
4    let ta = (qI, epsilon,ft.initialState) in
5    let t = Set.map (fun st -> createTransitionFA st ft.initialState) ft.
     acceptStates in
6    {
7      alphabet = ft.alphabet;
8      states = Set.add qI ft.states;
9      initialState = qI;
10     transitions = Set.add ta (Set.union t ft.transitions);
11     acceptStates = Set.add qI ft.acceptStates
12   }
13
```

6.4.4 Interseção

A interseção é a operação mais difícil de implementar. De maneira intuitiva, podemos perceber que a forma de ocorrer a interseção é percorrer os dois autómatos em simultâneo. Ao contrário da união e concatenação em que "ligamos" os autómatos respeitando os autómatos individuais, aqui queremos "juntar" os dois autómatos num só. Por isso, cada estado do novo autómato vai representar dois estados um de 'fa' e outro de 'fb'.

Ao estarmos num estado do novo autómato, estamos ao mesmo tempo num estado de 'fa' e num estado de 'fb'. É assim preciso combinar todos os estados de 'fa' com todos os estados de 'fb' juntos dois a dois. O conjunto de estados do novo autómato é o produto cartesiano dos estados de 'fa' e 'fb'.

O alfabeto no novo autómato é a interseção dos alfabetos de 'fa' e 'fb', porque todos os símbolos do alfabeto têm de pertencer a ambos os autómatos.

Para as transições temos, se α pertencente ao novo alfabeto e sendo a_1, a_2 estados pertencentes a 'fa' e b_1, b_2 estados pertencentes a 'fb' então

$$\langle a_1, b_1 \rangle \xrightarrow{\alpha} \langle a_2, b_2 \rangle \text{ apenas se } a_1 \xrightarrow{\alpha} a_2 \text{ e } b_1 \xrightarrow{\alpha} b_2$$

A função *findTransitions2* é chamada para cada par de estados, 'a' pertencente a 'fa' e o estado 'b' pertence 'fb'. A função vai verificar para cada símbolo do novo alfabeto se existem transições em 'fa' e 'fb' que saiam de 'a' e 'b' e quais os estados de chegada dessas transições. Com base na expressão teórica anterior vão ser construídas as transições novas dos autómatos.

Como também são admitidas transições vazias nos autómatos finitos vai ser feito o mesmo processo para se criar as transições do novo autómato. No entanto, como se pode adicionar uma transição vazia de um estado para ele próprio e isso não altera a lógica do autómato e o seguinte é possível

$$\langle a_1, b_1 \rangle \xrightarrow{\epsilon} \langle a_2, b_1 \rangle \text{ se } a_1 \xrightarrow{\epsilon} a_2$$

ou seja, se num par de estados o estado de 'fa' tem uma transição vazia para outro estado de 'fa', mas o estado de 'fb' não tem nenhuma transição vazia, o estado de 'fa' pode transitar para outro estado e o estado de 'fb' ficar igual.

Com a adição das transições vazias às restantes transições fica completo o conjunto de transições do novo autómato.

A função *makeName2* é usada para concatenar duas *strings*, juntando assim os nomes dos dois estados num nome só.

O resultado no final é o conjunto das transições do novo autómato em que o conjunto de estados, o estado inicial e os estados de aceitação são o produto cartesiano dos respetivos conjuntos em 'fa' e 'fa' e o alfabeto é a interseção dos alfabetos dos dois autómatos.

Listagem 6.7: Função de interseção de autómatos finitos *evalIntersectFA*.

```
1  let evalIntersectFA (fa: FiniteAutomaton.t) (fb: FiniteAutomaton.t):
     FiniteAutomaton.t =
2
3    let s = Set.product fa.states fb.states in
4    let newAlphabet = Set.inter fa.alphabet fb.alphabet in
5    let trans = Set.flat_map (fun (a,b) -> findTransitions2 a b fa.transitions
      fb.transitions newAlphabet ) s in
6    let newStates = Set.map (fun (a,b) -> makeName2 a b) s in
7    let s1 = Set.product fa.acceptStates fb.acceptStates in
8    let newAccept = Set.map (fun (a,b)  -> makeName2 a b) s1 in
9    {
10     alphabet = newAlphabet;
11     states =  newStates;
12     initialState = makeName2 fa.initialState fb.initialState;
13     transitions =  trans;
14     acceptStates = newAccept
15   }
16
```

6.5 Expressões regulares

As expressões regulares na biblioteca OCamlFLAT já têm definidas no seu tipo as três operações regulares: união (*Plus*), concatenação (*Seq*) e fecho de Kleene (*Star*). Por esta razão estas operações no módulo *Composition* podem ser feitas de forma direta, como se pode ver o exemplo da união na listagem 6.8.

A operação de interseção não sendo uma operação regular já não existe no módulo das expressões regulares. Por isso foi preciso arranjar uma solução para o problema, após alguma pesquisa foi verificado que a construção de uma interseção de expressões regulares seria difícil de se concretizar.

Como qualquer expressão regular é convertível em autómato finito e vice-versa, é feita a conversão das expressões e chamada a função de interseção de autómatos finitos, o resultado é convertido de volta para expressão regular.

Listagem 6.8: Função de união de expressões regulares *evalPlusRE*.

```
1  let evalPlusRE (ra: RegularExpression.t) (rb: RegularExpression.t) :
     RegularExpression.t =
2    Plus(ra, rb)
3
```

6.6 Gramáticas livre de contexto

Para as operações regulares, nas gramáticas livres de contexto é aplicada a lógica usada nos autómatos finitos, tendo em conta o formalismo das gramáticas. Por a operação de interseção não ser fechada nas linguagens independentes de contexto não foi aplicada a este modelo.

6.6.1 União

A união de gramáticas segue uma lógica semelhante à dos autómatos finitos. É criado um símbolo inicial novo e duas produções novas, que têm à cabeça o símbolo inicial novo e o corpo é os símbolos iniciais das duas gramáticas.

$$S_n \rightarrow S_a$$

$$S_n \rightarrow S_b$$

Assim a nova gramática tem como conjunto de símbolos terminais (chamado variáveis na biblioteca) a união dos conjuntos de símbolos terminais das gramáticas 'ca' e 'cb' com a adição do novo símbolo inicial. O conjunto de símbolos não terminais (chamado alfabeto na biblioteca) é a união dos conjuntos de símbolos não terminais de 'ca' e 'cb'. As produções da nova gramática são os conjuntos de todas as produções de 'ca' e 'cb' mais as produções novas criadas.

Listagem 6.9: Função de união de gramáticas livres de contexto *evalPlusCFG*.

```
1  let evalPlusCFG (ca: ContextFreeGrammarBasic.t) (cb: ContextFreeGrammarBasic
     .t) : ContextFreeGrammarBasic.t =
2
3     let open ContextFreeGrammarBasic in
4      let s =  str2symb (IdGenerator.gen("S")) in
5     let r1 = {head = s; body = [ca.initial]} in
6     let r2 = {head = s; body = [cb.initial]} in
7     {alphabet = Set.union ca.alphabet cb.alphabet;
8     variables = Set.add s (Set.union ca.variables cb.variables);
9     initial = s;
10    rules = Set.add r1 (Set.add r2 (Set.union ca.rules cb.rules))
11    }
12
```

6.6.2 Concatenação

A concatenação de duas gramáticas é a sequência de uma seguir a outra. Para alcançar esse efeito é criado um símbolo inicial novo e criada uma produção nova que tem como cabeça o símbolo novo e como corpo os símbolos iniciais das duas gramáticas em sequência.

$$S_n \rightarrow S_a S_b$$

Portanto, a nova gramática vai ser semelhante à gramática anterior feita por união, em que o conjunto de símbolos terminais, o conjunto de símbolos não terminais e o conjunto das produções vai ser a união dos respetivos componentes das gramáticas 'ca' e 'cb'. Sendo que vai ser adicionado aos símbolos não terminais o símbolo novo, e às produções a produção nova .

Listagem 6.10: Função de concatenação de gramáticas livres de contexto *evalSeqCFG*.

```
1  let evalSeqCFG (ca: ContextFreeGrammarBasic.t) (cb: ContextFreeGrammarBasic.
     t) : ContextFreeGrammarBasic.t =
2
3     let open ContextFreeGrammarBasic in
4      let s =  str2symb (IdGenerator.gen("S")) in
5     let r1 = {head = s; body = [ca.initial; cb.initial]} in
6     {alphabet = Set.union ca.alphabet cb.alphabet;
7     variables = Set.add s (Set.union ca.variables cb.variables);
8     initial = s;
9     rules =Set.add r1 (Set.union ca.rules cb.rules)
10    }
11
```

6.6.3 Fecho de Kleene

Como fecho Kleene admite a palavra vazia, é criado um estado inicial novo e uma produção cuja cabeça é o símbolo novo e o corpo é o símbolo vazio.

É também criada a produção que permite que a gramática seja reiterada n vezes. Esta produção tem o símbolo novo na cabeça, e o corpo é o símbolo novo seguido do símbolo inicial da gramática.

$$S_n \rightarrow S_n S_t$$

$$S_n \rightarrow \epsilon$$

A nova gramática vai ser semelhante à gramática 'ct' com exceção do símbolo inicial que é o símbolo novo. É adicionado o símbolo novo ao conjunto de símbolos não terminais, e as produções novas ao conjunto de produções.

Listagem 6.11: Função de fecho de Kleene de gramáticas livres de contexto *evalStarCFG*.

```
1  let evalStarCFG (ct: ContextFreeGrammarBasic.t)  : ContextFreeGrammarBasic.t
     =
2
3    let open ContextFreeGrammarBasic in
4     let s =  str2symb (IdGenerator.gen("S")) in
5    let r1 = {head = s; body = [s;ct.initial]} in
6    let r2 = {head = s; body = [epsilon]} in
7    {alphabet = ct.alphabet;
8    variables = Set.add s ct.variables;
9    initial = s;
10   rules = Set.add r1 (Set.add r2 ct.rules)
11   }
12
```

6.7 Autómatos de pilha

Nas operações regulares, a composição é feita de maneira semelhante aos autómatos finitos, mas tendo em consideração a existência da pilha nos autómatos de pilha. Tal como foi mencionado nas gramáticas livres de contexto a operação de interseção não é fechada nas linguagens independentes de contexto por isso não foi aplicada a este modelo.

Nas operações de composição dos autómatos de pilha prosopõe-se que estes têm como critério de aceitação os estados de aceitação. Por isso, no autómato resultante das operações de composição o critério de aceitação ocorre por estados de aceitação.

6.7.1 União

A união nos autómatos de pilha vai ter a mesma estrutura que os autómatos finitos, existe um estado inicial novo e duas transições que o "ligam" aos estados iniciais de cada autómato. A diferença vai estar na existência da pilha que vamos ter de ter em conta.

Para além do estado inicial novo criado, é também criado um símbolo inicial novo da pilha, cujo objetivo é apenas inicializar a pilha do novo autómato. São criadas duas transições vazias do estado novo para o estado inicial de cada autómato, 'pa' e 'pb'. Nestas é desempilhado o símbolo da pilha novo e empilhado o símbolo inicial da pilha dos autómatos respetivos.

Assim, o estado inicial e o símbolo inicial de pilha do novo autómato são o estado novo e o símbolo novo. O alfabeto de entrada e o alfabeto da pilha são ambos a união dos alfabetos de 'pa' e 'pb', sendo que o alfabeto da pilha vai ter acrescido o símbolo novo.

O conjunto de estados de aceitação é a união dos estados de aceitação de 'pa' e 'pb'. O conjunto de estados e o conjunto de transições é a união dos conjuntos dos dois autómatos apenas que adicionadas as transições novas e o estado novo.

Listagem 6.12: Função de união de autómatos de pilha *evalPlusPDA*.

```
1  let evalPlusPDA (pa: PushdownAutomaton.t) (pb: PushdownAutomaton.t) :
     PushdownAutomaton.t =
2
3    let qI = str2state (IdGenerator.gen("q")) in
4    let sI = str2symb (IdGenerator.gen("s")) in
5    let t1 = (qI,sI,epsilon,pa.initialState,[pa.initialStackSymbol]) in
6    let t2 = (qI,sI,epsilon,pb.initialState,[pb.initialStackSymbol]) in
7    {
8      inputAlphabet = Set.union pa.inputAlphabet pb.inputAlphabet;
9      stackAlphabet = Set.add sI (Set.union pa.stackAlphabet pb.stackAlphabet)
     ;
10     states = Set.add qI (Set.union pa.states pb.states);
11     initialState = qI;
12     initialStackSymbol = sI;
13     transitions = Set.add t2 (Set.add t1 (Set.union pa.transitions pb.
     transitions));
14     acceptStates = Set.union pa.acceptStates pb.acceptStates;
15     criteria = true
16   }
17
```

6.7.2 Concatenação

Para a concatenação é necessário "ligar" os estados finais do primeiro autómato,'pa', ao estado inicial do segundo, 'pb'. Por isso é criada uma função auxiliar *createTransitionPDA*.

A função "liga" um estado final de 'pa' ao estado inicial de 'pb'. Como não se sabe com que símbolo de pilha 'pa' acaba, é necessário para cada símbolo da pilha de 'pa' criar uma transição vazia que desempilha o símbolo e empilha o símbolo inicial da pilha de 'pb'.

O alfabeto de entrada, o alfabeto da pilha e o conjunto de estados do novo autómato são a união dos conjuntos de 'pa' e 'pb'. O conjunto de transições é a união dos conjuntos de transições dos autómatos, acrescido do conjunto de transições novas criadas pela função

auxiliar. O estado inicial e o símbolo inicial da pilha são os mesmos de 'pa' e o conjunto de estados de aceitação é o conjunto de estados de aceitação de 'pb'.

Listagem 6.13: Função de concatenação de autómatos de pilha *evalSeqPDA*.

```
1  let evalSeqPDA (pa: PushdownAutomaton.t) (pb: PushdownAutomaton.t) :
     PushdownAutomaton.t =
2
3    let t = Set.flatten (Set.map (fun st -> createTransitionPDA st pb.
     initialState pb.initialStackSymbol pa.stackAlphabet) pa.acceptStates) in
4    {
5      inputAlphabet = Set.union pa.inputAlphabet pb.inputAlphabet;
6      stackAlphabet = Set.union pa.stackAlphabet pb.stackAlphabet;
7      states = Set.union pa.states pb.states;
8      initialState = pa.initialState;
9      initialStackSymbol = pa.initialStackSymbol;
10     transitions = Set.union t (Set.union pa.transitions pb.transitions);
11     acceptStates = pb.acceptStates;
12     criteria = true
13   }
14
```

6.7.3 Fecho de Kleene

No fecho de Kleene para cumprir com a aceitação da palavra vazia é criado um estado inicial novo, um símbolo inicial de pilha novo e uma transição nova. A transição é uma transição vazia do estado novo para o estado inicial do autómato e desempilha o símbolo da pilha novo e empilha o símbolo inicial da pilha do autómato.

A função auxiliar *createTransitionPDA* que é usada para a concatenação é também usada aqui para "ligar" os estados de aceitação ao estado inicial. É criado assim um conjunto de transições vazias entre os dois estados que desempilha qualquer símbolo da pilha de 'pt' e empilha o estado inicial.

O alfabeto de entrada do novo autómato é igual ao alfabeto de entrada de 'pt'. O símbolo inicial da pilha e o estado inicial são o símbolo novo e o estado novo respetivamente. O alfabeto da pilha é igual ao de 'pt' com a adição do símbolo novo. Também as transições e os estados de aceitação são iguais a 'pt', sendo que nas transições são adicionadas as transições novas, e o estado novo passa a fazer parte dos estados de aceitação.

Listagem 6.14: Função de fecho de Kleene de autómatos de pilha *evalStarPDA*.

```
1  let evalStarPDA (pt: PushdownAutomaton.t)  : PushdownAutomaton.t =
2
3    let qI = str2state (IdGenerator.gen("q")) in
4    let sI = str2symb (IdGenerator.gen("s")) in
5    let t1 = (qI,sI,epsilon,pt.initialState,[pt.initialStackSymbol]) in
6    let t = Set.flatten (Set.map (fun st -> createTransitionPDA st pt.
     initialState pt.initialStackSymbol pt.stackAlphabet) pt.acceptStates) in
7    {
8      inputAlphabet = pt.inputAlphabet;
9      stackAlphabet = Set.add sI pt.stackAlphabet;
10     states = Set.add qI pt.states;
11     initialState = qI;
12     initialStackSymbol = sI;
13     transitions = Set.add t1 (Set.union t pt.transitions);
14     acceptStates = Set.add qI pt.acceptStates;
15     criteria = true
16   }
17
```

6.8 Máquinas de Turing

As operações regulares nas máquinas de Turing são feitas de maneira semelhante aos autómatos, especialmente aos autómatos de pilha devido à existência da fita. Importante referir que a máquina de Turing inicializa a cabeça da fita à esquerda da palavra e em cada transição a cabeça da fita anda uma casa para a direita ou para esquerda, nunca ficando parada na mesma casa.

A operação de interseção ocorre de maneira diferente, devido à intricação da operação foi preciso a utilização de máquinas de Turing multifita, neste caso vão ser utilizadas duas fitas. Como a adição foi feita ao longo do desenvolvimento desta dissertação não foi possível ligar esta operação ao resto do código das composições.

Em todas as operações a nova máquina de Turing vai ter como o critério de aceitação os estados de aceitação, ou seja o valor do critério é verdadeiro. Como é assumido que estas operações são apenas chamadas para as máquinas de Turing que não são linearmente limitadas os marcadores são vazios para todas as máquinas por isso são metidos como igual aos marcadores 'ta', com exceção do fecho de Kleene em que são iguais a 'tt'. Aqui o símbolo B(branco) é representado pelo *empty*.

6.8.1 União

Nas máquinas de Turing tal como nos restantes autómatos é criado um estado inicial novo e um conjunto de transições que "ligam" o estado novo aos estados iniciais das máquinas de Turing, neste caso em vez de uma ligação por modelo vão ser precisas duas ligações. Para "ligar" o novo estado ao estado inicial de 'ta' é criada uma transição entre os estados

que lê branco e escreve branco na fita e faz um movimento para a esquerda ('L'). Para a máquina de Turing ficar com a cabeça à esquerda da palavra quando começar a iteração de 'ta' é criada outra transição do estado inicial de 'ta' para ele próprio que também é uma transição vazia, portanto lê e escreve branco e anda uma casa para a direita. São criadas transições da mesma maneira para 'tb'.

O alfabeto de entrada e o alfabeto da fita são ambos a união dos alfabetos de 'ta' e 'tb', tal como o conjunto dos estados de aceitação é a união dos estados de aceitação das duas máquinas. O conjunto dos estados é a união dos estados de 'ta' e 'tb' acrescidos do estado novo que passa a ser o estado inicial. À união dos conjuntos de transições de 'ta' e 'tb' adicionam-se as transições novas criadas para formar o conjunto de transições da nova máquina de Turing.

Listagem 6.15: Função de união de máquinas de Turing *evalPlusTM*.

```
1     let evalPlusTM (ta: TuringMachine.t) (tb: TuringMachine.t): TuringMachine.
      t =
2
3     let qI = str2state (IdGenerator.gen("q")) in
4     let t1 = (qI,empty,ta.initialState,empty,L) in
5     let t2 = (ta.initialState,empty,ta.initialState,empty,R) in
6     let t3 = (qI,empty,tb.initialState,empty,L) in
7     let t4 = (tb.initialState,empty,tb.initialState,empty,R) in
8     let t = Set.make [t1;t2;t3;t4] in
9     {
10      entryAlphabet = Set.union ta.entryAlphabet tb.entryAlphabet;
11      tapeAlphabet = Set.add empty (Set.union ta.tapeAlphabet tb.tapeAlphabet)
      ;
12      empty = empty;
13      states = Set.add qI (Set.union ta.states tb.states);
14      initialState = qI;
15      transitions = Set.union t (Set.union ta.transitions tb.transitions);
16      acceptStates = Set.union ta.acceptStates tb.acceptStates;
17      criteria = true;
18      markers = ta.markers
19    }
20
```

6.8.2 Concatenação

A concatenação nas máquinas de Turing é semelhante aos autómatos de pilha. É necessário "ligar" os estados finais de 'ta' ao estado inicial de 'tb', para isso é criada uma função auxiliar, *createTransitionTM*.

A função "liga" um estado final de 'ta' ao estado inicial de 'tb'. Para isso, por cada símbolo do alfabeto da fita é criada uma transição que lê o símbolo e volta-o a escrever, andando uma casa para a direita. Esta função é chamada para todos os estados de aceitação de 'ta'.

O alfabeto de entrada, o alfabeto da fita e o conjunto de estados são a união dos respetivos conjuntos em 'ta' e 'tb'. É preciso adicionar o branco ao alfabeto da fita no caso de nem 'ta' nem 'tb' o terem.

O estado inicial é o estado inicial de 'ta' e os estados de aceitação são os de 'tb'. O conjunto de transições é a união das transições de 'ta' e 'tb' com a adição das transições novas criadas.

Listagem 6.16: Função de concatenação de máquinas de Turing *evalSeqTM*.

```
1  let evalSeqTM (ta: TuringMachine.t) (tb: TuringMachine.t): TuringMachine.t =
2
3  let evalSeqTM (ta: TuringMachine.t) (tb: TuringMachine.t): TuringMachine.t =
4
5    let t = Set.flatten (Set.map (fun st -> createTransitionTM st tb.
     initialState ta.tapeAlphabet) ta.acceptStates) in
6    {
7      entryAlphabet = Set.union ta.entryAlphabet tb.entryAlphabet;
8      tapeAlphabet = Set.add empty (Set.union ta.tapeAlphabet tb.tapeAlphabet)
     ;
9      empty = empty;
10     states = Set.union ta.states tb.states;
11     initialState = ta.initialState;
12     transitions = Set.union t (Set.union ta.transitions tb.transitions);
13     acceptStates = tb.acceptStates;
14     criteria = true;
15     markers = ta.markers
16   }
17
```

6.8.3 Fecho de Kleene

No fecho de Kleene é necessário tal como no resto dos autómatos a criação de um estado inicial novo para "ligar" ao estado inicial de 'tt' com uma transição vazia com a diferença que nas máquinas de Turing são precisas duas transições como à semelhança da união, uma transição para mudar de estado que lê o branco e escreve o branco, movendo-se para a esquerda e outra que fica no estado inicial de 'tt' mas anda com a cabeça da fita para a direita. O novo estado inicial faz parte dos estados de aceitação.

Para criar as transições dos estados finais para o estado inicial 'tt' é usada a mesma função auxiliar, *createTransitionTM*, que na concatenação que vai receber um estado de aceitação de 'tt', o estado inicial de 'tt' e o alfabeto da fita. Para cada símbolo do alfabeto da fita é criada uma transição que lê e escreve o símbolo na fita e anda uma casa para a direita. A função é chamada para todos os estados de aceitação de 'tt'.

O alfabeto de entrada e o alfabeto da fita são iguais aos alfabetos de 'tt'. O conjunto de estados e conjunto de estados de aceitação são iguais aos de 'tt' com a adição em ambos do novo estado, que é também o estado inicial da nova máquina de Turing. Ao conjunto de

transições de 'tt' acresce as transições novas criadas para formar o conjunto de transições da nova máquina.

Listagem 6.17: Função de fecho de Kleene de máquinas de Turing *evalStarTM*.

```
1  let evalStarTM (tt: TuringMachine.t) : TuringMachine.t =
2    let qI = str2state (IdGenerator.gen("q")) in
3    let t1 = (qI,empty,tt.initialState,empty,L) in
4    let t2 = (tt.initialState,empty,tt.initialState,empty,R) in
5     let t = Set.flatten (Set.map (fun st -> createTransitionTM st tt.
     initialState tt.tapeAlphabet) tt.acceptStates) in
6
7     {
8       entryAlphabet = tt.entryAlphabet;
9       tapeAlphabet = Set.add empty tt.tapeAlphabet;
10      empty = empty;
11      states = Set.add qI tt.states;
12      initialState = qI;
13      transitions = Set.add t2 (Set.add t1 (Set.union t tt.transitions));
14      acceptStates = Set.add qI tt.acceptStates;
15      criteria = true;
16      markers = tt.markers
17    }
18
```

6.8.4 Interseção

A interseção das máquinas de Turing funciona com duas fitas em vez de uma como nas outras operações. Esta operação funciona de maneira semelhante à interseção nos autómatos, mas de maneira mais simples, pois existem duas fitas em paralelo.

Os estados da nova máquina vão representar cada um dois estados um de 'ta' e outro de 'tb', para isso tal como nos autómatos finitos também aqui o novo conjunto de estados vai ser o produto cartesiano dos estados de 'ta' e 'tb'. Após ser criada a lista de pares de estados, para cada par é chamada a função auxiliar *createsTransitionsInter*.

A função auxiliar vai receber um estado de 'ta', um estado de 'tb' e os conjuntos de transições de 'ta' e 'tb'. Em cada um dos conjuntos de transições vão ser filtradas as transições que têm como estado de partida o estado de 'fa' e o estado de 'fb'. É de seguida feito o produto cartesiano das transições de 'fa' e de 'fb'. Cada par de transições é transformado numa só transição.

Como a nova máquina vai ser uma máquina de duas fitas, nas transições o símbolo que se lê na fita, o símbolo que se escreve na fita e a direção da cabeça de leitura passam a ser representados por uma lista em que a ordem em que se encontram corresponde à fita que pertencem. Assim, quando é para juntar as duas transições numa só, metem-se os elementos da transição de 'ta' todos na primeira posição e os de 'tb' na segunda posição. A primeira fita corresponde a 'ta' e a segunda a 'tb'.

Fica assim criado o conjunto de transições da nova máquina de Turing multifita. O alfabeto de entrada vai ser a interseção dos alfabetos de entrada de 'ta' e 'tb', o alfabeto da fita vai ser a união dos alfabetos das fitas das duas máquinas. O conjunto de estados e os estados de aceitação são o produto cartesiano dos respetivos conjuntos das duas máquinas e o estado inicial é o estado inicial de 'ta' e de 'tb' juntos.

Listagem 6.18: Função de interseção de máquinas de Turing *evalIntersectTM*.

```
1   let evalIntersectTM (ta: TuringMachine.t) (tb: TuringMachine.t):
      TurMachMultiTypes.t =
2
3     let s = Set.product ta.states tb.states in
4     let newEntryAlphabet = Set.inter ta.entryAlphabet tb.entryAlphabet in
5     let trans = Set.flat_map (fun (a,b) -> createsTransitionsInter a b ta.
      transitions tb.transitions ) s in
6     let newStates = Set.map (fun (a,b) -> makeName2 a b) s in
7     let acceptS = Set.product ta.acceptStates tb.acceptStates in
8     let newAccept = Set.map (fun (a,b)  -> makeName2 a b) acceptS in
9     {
10      entryAlphabet = newEntryAlphabet;
11      tapeAlphabet = Set.union ta.tapeAlphabet tb.tapeAlphabet;
12      empty = empty;
13      states = newStates;
14      initialState = makeName2 ta.initialState tb.initialState;
15      transitions = trans;
16      acceptStates = newAccept;
17      criteria = true; (* true = acceptStates | false = stop *)
18      markers = ta.markers
19    }
20
```

6.9 Redução de modelos compostos

Reduzir uma composição de modelos a um modelo atómico corresponde à operação mais importante e mais ambiciosa deste trabalho. Na verdade, essa operação acaba por se desdobrar em cinco variantes porque é preciso indicar o tipo do modelo atómico pretendido. Os nomes das funções que implementam as cinco variantes são: *evalFA*, *evalRE*, *evalCFG*, *evalPDA* e *evalTM*.

Existem três problemas principais que é preciso superar na definição das reduções. Primeiro problema: Uma composição pode ser mista, e por isso é preciso executar um passo inicial que converte todos os modelos integrantes da composição para o tipo pretendido para o resultado. Obtida uma composição homogénea, já se poderão aplicar as operações de composição definidas nas secções anteriores. Segundo problema: Relativamente às conversões atrás referidas, a teoria diz que algumas são impossíveis, e nesse caso desiste-se de realizar a redução (é gerada uma exceção). Terceiro problema: Podem existir conflitos de nomes entre os vários modelos da composição homogénea, por isso é preciso ainda

inserir um passo de renomeação, usando nomes únicos, de todos os estados e símbolos não terminais de cada modelo constituinte.

Uma ligeira complicação técnica é que a biblioteca suporta três vistas distintas para cada tipo de modelo: (1) Vista como valor, baseada numa representação direta usando tipo estrutural; (2) Vista como objeto, ligada a uma organização dos conceitos usando classes e herança; (3) Vista como nome pertencente ao repositório. As duas últimas vistas impõem um passo inicial adicional que converte todos os objetos e todos os nomes de uma composição para as representações correspondentes.

Após esta introdução, esta secção vai descrever com detalhe todo o que está envolvido nas operações de redução.

Listagem 6.19: Função *evalFA*.

```
1  let evalFA (c: t) : FiniteAutomaton.t  =
2    let c1 = transformt c in
3    match isFAConvertivel c1 with
4      | true -> evalMixFA c1
5      | false -> Error.fatal "evalFA"
6
```

A redução da composição começa pela invocação da função *transformt* para converter todos os modelos integrantes que estão na forma de objeto ou de nome para o formato de representação direta de um modelo.

Como para as operações de composição, por exemplo a união, têm de ser aplicadas para composições onde os componentes têm todos o mesmo o tipo, o seguinte passo é ver se todos os modelos que compõem a composição são convertíveis para o tipo da composição que queremos. Na função *evalFA* vai ser chamada a função *isFAConvertivel*, na função *evalRE* vai ser chamada a função *isREConvertivel* e para os restantes modelos aplica-se o mesmo. Estas funções são recursivas e seguem a estrutura recursiva da composição.

A função *isTMConvertivel* devolve verdadeiro para qualquer um dos cinco modelos enquanto as funções *isPDAConvertivel* e *isCFGConvertivel* retornam verdadeiro para todos exceto a máquina de Turing. As funções *isREConvertivel* e *isFAConvertivel* devolvem verdadeiro para autómatos finitos e expressões regulares. Para gramáticas livres de contexto vai se testar se a gramática é regular, para os autómatos de pilha vai se testar se o autómato pode ser um autómato finito, finalmente para as máquinas de Turing é testado se a máquina pode ser um autómato finito.

Após se fazer se verificar se são convertíveis verifica-se o resultado, se for falso é lançado um erro, se for verdadeiro vai ser chamada a função *evalMix*. A função *evalFA* chama a função *evalMixFA*, a função *evalRE* chama a função *evalMixRE* e as restantes funções *eval* chamam as funções *evalMix* respetivas.

Listagem 6.20: Função *evalMixFA*.

```
1  let evalMixFA (c:t) : FiniteAutomaton.t =
2    let c1  = comp2facomp c in
3    let c2 = rename c1 in
4    calcEvalFA c2
5
```

As funções *evalMix* recebem uma composição de representações que são todas possíveis de ser convertidas no tipo da função. Tal como para as funções *eval* também vão existir cinco funções *evalMix*, *evalMixFA*, *evalMixRE*, *evalMixCFG*, *evalMixPDA* e *evalMixTM*. Elas têm a mesma estrutura com uma ligeira diferença na função *evalMixRE*.

A função recebe uma composição 'c' que pode ser composta por várias representações diferentes e vai converter numa composição com apenas um tipo de representações. A função *evalMixFA* chama a função *comp2facomp*, a função *evalMixRE* chama a função *comp2recomp* e as restantes funções *evalMix* chamam as respetivas funções de conversão. As funções de conversão são funções recursivas que chamam as funções de conversão do módulo *PolyBasic*. As funções *comp2CFGcomp* e *comp2PDAcomp* não apresentam a conversão no caso da máquina de Turing, pois estas não são convertíveis para gramáticas livres de contexto e autómatos de pilha. A conversão de máquinas de Turing para autómatos finitos é possível através da função *downgradeModelToFiniteAutomaton* do módulo das *TuringMachine*, nas expressões regulares é chamada a mesma função *downgradeModelToFiniteAutomaton* e depois realizada a conversão para expressão regular.

Listagem 6.21: Função *comp2facomp*.

```
1     let rec comp2facomp (c: t): t =
2     match c with
3       | Plus (a,b) ->
4           Plus (comp2facomp a, comp2facomp b)
5       | Seq (a,b) ->
6           Seq (comp2facomp a, comp2facomp b)
7       | Intersect (a,b) ->
8           Intersect (comp2facomp a, comp2facomp b)
9       | Star t ->
10          Star (comp2facomp t)
11      | FA fa ->
12          FA fa
13      | RE re ->
14          FA (PolyBasic.re2fa re)
15      | CFG cfg ->
16        FA (PolyBasic.cfg2fa cfg)
17      | PDA pda ->
18        FA (PolyBasic.pda2fa pda)
19      | TM tm ->
20        let tmobj = new TuringMachine.model (Representation tm) in
21        FA (tmobj#downgradeModelToFiniteAutomaton)#representation
22      | _ -> Error.fatal "comp2facomp"
23
```

Depois de uma haver uma composição cujos elementos sejam todos do mesmo tipo é chamada a função *rename* que faz a renomeação dos estados dos autómatos finitos, autómatos de pilha e máquinas de Turing e das variáveis não terminais das gramáticas livres de contexto, metendo um número único à frente do nome destes elementos. Para as expressões regulares não é necessário esta função.

Após estes preparativos todos é assim possível chamar as funções *calcEval* que vão proceder à redução das composições, estas são recursivas e seguem a recursividade da estrutura das composições.

6.10 Repositório

A biblioteca mantém um repositório que permite associar nomes a composições existentes. No fundo, não é mais que um dicionário. Note que no repositório também é possível registar modelos atómicos porque um modelo atómico é um caso particular do modelo composto.

O repositório serve uma necessidade da parte gráfica: todos os modelos que integram uma composição têm de ter um nome, para a composição poder ser visualizada no ecrã.

Guarda uma variável global que é uma lista de pares cuja primeira coordenada é o nome da composição e a segunda coordenada é a composição.

No repositório é possível saber se uma composição já existe, apagar elementos da lista, fazer o update de uma determinada composição e ir buscar uma composição à lista. Existe ainda a função que devolve as composições sobre formato de texto.

7

Suporte para composição de modelos da plataforma OFLAT

No que diz respeito à aplicação gráfica, este trabalho consistiu em adicionar suporte para visualização e manipulação das composições (modelos compostos). Logo de início, foi preciso estudar o trabalho já feito para os cinco tipos de modelos atómicos já suportados. Seguidamente, a integração das composições foi efetuada respeitando o mais possível as soluções já existentes no código. Essas soluções refletem um conjunto de opções concretas sobre a forma de realizar o modelo MVC.

Um segundo ponto que precisa de ser destacado é que a forma de processar e de apresentar visualmente as composições é fortemente inspirada no trabalho já feito para as expressões regulares. Isso é perfeitamente natural porque os conceitos de expressão regular e de composição são aparentados — uma composição é uma espécie de expressão regular sobre modelos FLAT.

Contudo, surgiram duas questões novas: a necessidade de permitir nomear os novos modelos criados pelo utilizador (para poderem ser referidos nas composições), e a necessidade de poder examinar os modelos atómicos constituintes duma composição.

7.1 MVC

A plataforma OFLAT usa a arquitetura Model-View-Control(MVC), cujo objetivo é dividir o sistema em três componentes, cada uma com responsabilidades diferentes.

No caso do nosso sistema, cada modelo FLAT ativo dispõe das três componentes do modelo MVC. De notar que o mesmo se aplica às composições, que são vistas como um modelo especial. Para cada modelo, a componente modelo diz respeito à biblioteca OCamlFLAT, onde estão implementadas as operações lógicas do modelo; as componentes visualizador e controlador estão implementadas na aplicação gráfica, sendo que a visualizador implementa todos os aspetos visuais e o controlador processa os inputs dos utilizadores, fazendo pedidos ao modelo e dando instruções ao visualizador.

Normalmente o ecrã da aplicação mostra apenas um modelo FLAT ativo com o qual o

utilizador interage. Mas existem algumas operações em que o resultado é suficientemente intricado para precisar de ser visualizado num segundo painel. Nesse caso, o ecrã da aplicação é dividido ao meio em dois painéis, ficando o modelo ativo no painel da esquerda e o resultado no painel da direita. Por vezes o resultado é um novo modelo FLAT mas nesse caso considera-se que o modelo está inativo porque não estão disponíveis as suas operações habituais. O objetivo é apenas a visualização do resultado.

Para lidar com os dois painéis internamente são mantidas duas variáveis imperativas que guardam dois controladores, o primeiro associado ao painel da esquerda e o segundo associado ao painel da direita.

As composições aproveitam os dois painéis da seguinte forma: a composição ativa está disponível no painel da esquerda e o utilizador pode clicar no nome de qualquer dos modelos constitutivos para esse modelo aparecer no painel da direita e poder ser visto pelo utilizador.

7.2 Interface da composição de modelos

A composição de modelos é apresentada usando a descrição textual da composição que correspondente à árvore estrutural, à semelhança das expressões regulares, como apresentado na figura 7.1.

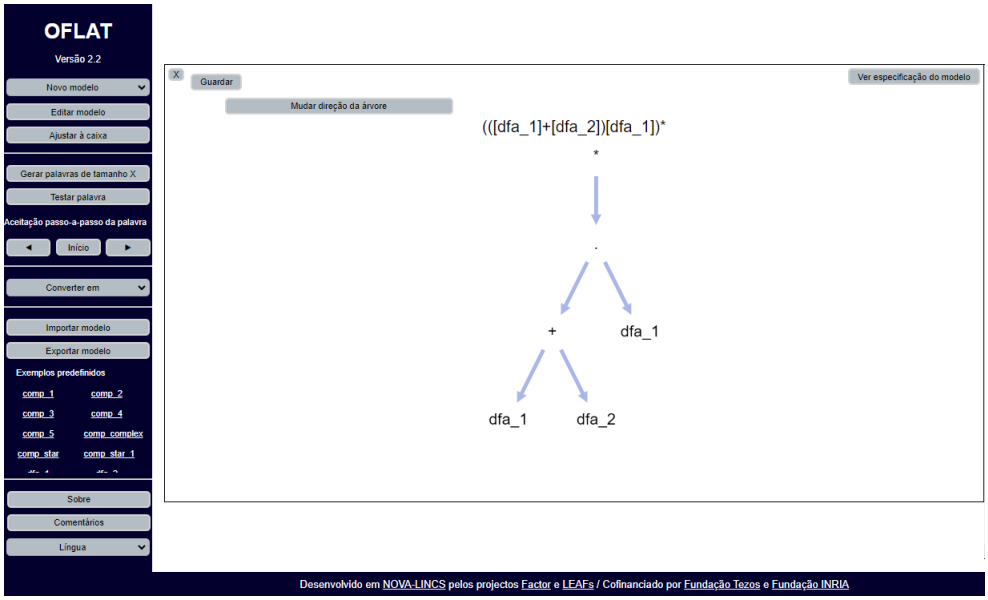

Figura 7.1: Apresentação da composição na plataforma OFLAT

A ideia de ter a composição de modelos a aparecer em formato de árvore foi emprestada das expressões regulares. Pois, uma composição é caracterizada por uma estrutura arbórea de maneira semelhante às expressões regulares. Se o utilizador quiser examinar os vários modelos que fazem parte da composição, ao clicar num modelo da composição na árvore o ecrã divide-se em dois e no ecrã da direita aparece a representação do modelo selecionado (figura 7.2).

Quando se cria um controlador para um modelo é preciso passar um argumento booleano que vai ditar se o controlador é instalado ocupando todo o ecrã, ou se o ecrã é dividido ao meio sendo o controlador instalado no painel da direita.

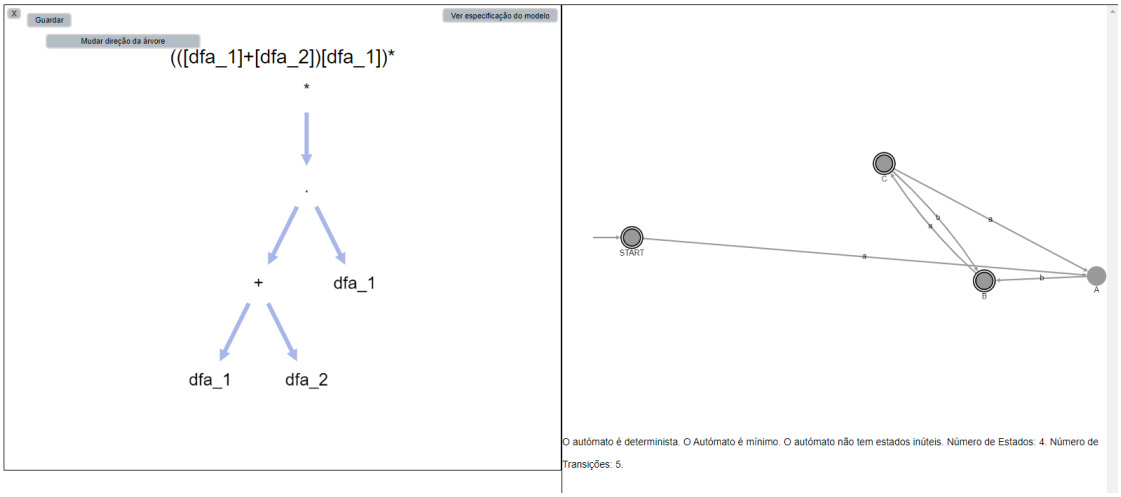

Figura 7.2: Exemplo do modelo aparecer à direita ao clicar no nó

A construção da árvore de visualização foi feita no módulo novo criado, o *CompositionView*, e segue a mesma estrutura que as expressões regulares. A função de construção navega de maneira recursiva pela composição e vai criando nós e arcos usando as operações da biblioteca *Cytoscape.js*.

Uma diferença relativamente às expressões regulares, é que as folhas da árvore são reativas (como foi explicado atrás) e isso implicou um tratamento especial, que requereu alguma investigação para se descobrir uma boa solução técnica. Foi necessário instalar um listener nas folhas da árvore. Os listeners são funções guardadas no controlador que são chamadas quando existe uma interação na interface por parte do utilizador.

É possível criar uma nova composição usando o mesmo esquema de código já existente para as expressões regulares. Encontra-se no elemento HTML select como o nome "Novo Modelo" a opção de composição. O utilizador deve escrever a descrição textual da composição. Esta vai ser usada pelo controlador para fazer a alteração do painel. Todo o que se apresenta no painel desaparece e aparece o novo modelo.

A operação de edição segue a mesma estrutura interna que a criação da composição. O utilizador deve primeiro selecionar o modelo de composição que quer alterar e só depois ativar o botão "editar modelo". O controlador mostra a descrição textual da composição que o utilizador pode alterar.

Existe ainda o botão "Guardar" que ao ser ativado guarda no repositório o modelo com o nome que o utilizador lhe quiser dar. O botão chama um listener do controlador que envia o modelo presente no painel para a biblioteca OCamlFLAT onde vai ser guardado no módulo do repositório.

7.3 *Accept* e *Generate*

As funcionalidades de reconhecimento de uma palavra (*accept*) e de geração de palavras (*generate*) são funcionalidades que estão disponíveis na interface gráfica. Ao ativar o botão "Testar palavra" indica-se a palavra que a ser testada, e no caso da composição a resposta aparece num diálogo. Quando é ativado o botão do "Gerar palavras de tamanho x" indica-se o tamanho máximo das palavras, e é aberto o painel mais à direita com as palavras geradas.

A implementação destas duas funções é uniforme para todos os modelos e obedece a um protocolo interno que se baseia na existência de duas funções: o *accept* e o *getWords*, no controlador do modelo. Portanto, bastou adicionar ao controlador as duas operações para ficar todo funcional. Cada uma destas funções chama a sua função respetiva da biblioteca, o *accept* chama o *accept*, e o *getWords* chama o *generate*.

7.4 Conversão

Para se fazer a conversão das composições usa-se o mesmo esquema de código que já existia na plataforma. As conversões usam um protocolo interno que teve de ser estudado para poder ser adicionada as operações de conversão a todos os modelos da plataforma. Para fazer as conversões funcionarem corretamente foi preciso fazer ajustes e alterações em algumas partes do código, algumas destas mais trabalhosas de ultrapassar.

Na interface da do OFLAT é possível escolher a composição na barra lateral azul escura à esquerda onde existe um select chamado "converter em" onde se pode escolher o modelo. Neste trabalho foi adicionada a opção "Máquina de Turing" que ainda não existia, que por erro ainda não aparecia esta opção. (figura 7.3).

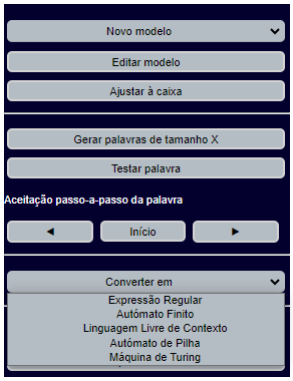

Figura 7.3: Select de conversão

As operações de conversão já se encontravam parcialmente implementadas na plataforma e seguiam um protocolo interno.

Os métodos de conversão são implementados no controlador da composição, o *CompositionController*, e devolvem um visualizador. Existe um método para cada modelo atómico suportado pela plataforma: o *convertToRegExp* é chamado quando se clica na

opção das expressões regulares, o *convertToFA* quando se escolhe autómatos finitos, o *convertToCFG* quando se escolhe gramáticas livres de contexto, o *convertToPDA* quando se escolhe autómatos de pilha e o *convertToTM* quando se escolhe máquinas de Turing.

Estes métodos têm todas a mesma estrutura. Por exemplo, o *convertToFA* (método que converte uma composição num autómato finito) chama a função *evalFA* da biblioteca, e é criado um visualizador com o autómato finito recebido da biblioteca.

Listagem 7.1: Método que implementa *convertToFA* no controlador da composição.

```
1     method convertToFA =
2       let open FiniteAutomatonView in
3       let fa = comp1#evalFA in
4         new FiniteAutomatonView.model (Representation fa)
5
```

O controlador de cada modelo recebe o visualizador criado e depois inicializa o modelo do lado direito do ecrã.

8

Avaliação e trabalho futuro

8.1 Avaliação

O conjunto das funções *eval* foi testado de maneira bastante exaustiva quando foi implementada a parte gráfica. Foi possível criar várias composições homogéneas e mistas. A criação de composições mistas permite verificar se as verificações de conversão estão a funcionar corretamente. Durante os testes, as composições mais pequenas foram as mais testadas, porque os resultados tendem a ser relativamente compreensíveis. Já composições mais complexas dão facilmente origem a resultados muito complicados e difíceis de perceber. Mesmo assim, há formas indiretas de ganhar alguma confiança relativamente a resultados complexos, como por exemplo mandar gerar todas as palavras com um dado tamanho máximo, e comparar.

Houve alguns testes interessantes que puderam ser feitos, como por exemplo a união de um autómato com ele mesmo. Quando é feita a redução para um autómato finito atómico, o resultado costuma ser bastante confuso. No entanto, através da minimização consegue-se confirmar que se obtém o mesmo autómato.

Durante os testes algo que foi possível perceber com o facto de os autómatos finitos terem uma funcionalidade para tornar determinista e minimizar é que o autómato resultante da composição pode ser em quase todos os casos simplificado. Nos restantes modelos não foi possível fazer esta simplificação.

Não foram feitos testes para determinar os limites do sistema. No entanto, podemos observar que alguns testes feitos com expressões regulares faziam com que o OFLAT ficasse bloqueado, pelo facto de serem geradas expressões tão grandes que causavam *overflow* de memória. Uma defeciência conhecida do sistema é que não têm um bom simplificador de expressões regulares.

Durante a dissertação houve o supervisionamento do orientador, que tendo lecionado a cadeira de Teoria de Computação, pode dar feedback sobre o funcionamento da plataforma OFLAT enquanto ferramenta educativa.

8.2 Trabalho Futuro

A composição de modelos é um tema vasto, que pode ser expandido de diversas maneiras no futuro. Quer com novas operações de composição ou novos modelos.

Existem várias operações de composição que podem ser adicionadas e algumas até chegaram a ser pensadas para esta dissertação, por exemplo, complemento e o inverso.

Para além da adição de operações de composição, também pode ser interessante acrescentar modelos novos à biblioteca OCamlFLAT e acrescentá-los à composição de modelos. Alguns exemplos são as gramáticas dependentes de contexto e as gramáticas irrestritas apresentadas no capítulo da teoria.

Estas adições à biblioteca devem ser acompanhadas do seu suporte na plataforma OFLAT. Na interface gráfica podem ser sempre adicionadas novas funcionalidades para tornar a interação com o modelo composto mais agradável. Por exemplo, poder editar o modelo composto diretamente na árvore estrutural em vez de na representação escrita.

8.3 Conclusão

Nesta dissertação foi adicionado suporte para a composição de modelos na biblioteca OCamlFLAT e na plataforma OFLAT. A composição de modelos foi trabalhada sobre os modelos já existentes nas biblioteca e plataforma.

Na biblioteca OCamlFLAT foi adicionado um módulo novo que suporta as operações regulares (união, concatenação e fecho de Kleene) para os cinco modelos já suportados pela biblioteca e ainda a operação de interseção para os modelos para os quais é fechada. Foi preciso haver primeiro estudo teórico dos modelos e das operações de composição e depois um conhecimento da biblioteca, para assim serem concretizadas as ideias teóricas.

Para além das operações de composição em si, foi preciso arranjar um esquema de código que suporta-se as composições mistas(composições com pelo menos dois tipos diferentes de modelos).

Na plataforma OFLAT foi adicionado o suporte e manipulação para a visualização de modelos compostos. Apesar de muitas das funcionalidades já terem protocolos e esquemas de código implementados, foi preciso ajustar o código para suportar noção de composição. Para isso foi preciso compreender como estava estruturada a plataforma OFLAT e o código das funcionalidades. Resolver algumas lacunas que existiam no código, algumas trabalhosas. E expandir algumas funcionalidades.

Foi possível ainda adicionar um repositório onde o utilizador pode guardar modelos.

Bibliografia

[1] *Example of unrestricted grammar which produces non-context-sensitive language*. URL: https://cs.stackexchange.com/questions/68533/example-of-unrestricted-grammar-which-produces-non-context-sensitive-language (acedido em 2024-07-15) (ver p. 13).

[2] *FAdo documentation*. URL: https://www.dcc.fc.up.pt/~rvr/FAdoDoc/comboperations.html#module-FAdo.comboperations (acedido em 2023-07-08) (ver p. 19).

[3] *FAdo: tools for Formal Languages manipulation*. URL: https://fado.dcc.fc.up.pt/ (acedido em 2023-07-08) (ver p. 19).

[4] *Finite Automata Tool*. URL: http://cl-informatik.uibk.ac.at/software/fat/index.php (acedido em 2023-07-09) (ver p. 21).

[5] C. M. M. Fritas. *Autómatos de pilha nas ferramentas OCaml-FLAT/OFLAT*. Março, 2023 (ver p. 24).

[6] J. E. Hopcroft, R. Motwani e J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3ª ed. 2007 (ver p. 15).

[7] *JFLAP Version 7.1*. URL: https://www.jflap.org/ (acedido em 2023-07-08) (ver p. 17).

[8] *JFLAP Version 7.1 - Building A Turing Machine*. URL: https://www.jflap.org/tutorial/turing/one/index.html#block (acedido em 2023-07-11) (ver p. 17).

[9] *js_of_ocaml manual*. URL: https://ocsigen.org/js_of_ocaml/latest/manual/library (acedido em 2023-07-11) (ver p. 28).

[10] S. C. Kleene. *Representation of Events in Nerve Nets and Finite Automata*. 1951 (ver p. 4).

[11] P. Linz. *An introduction to formal languages and automata*. Jones & Bartlett Learning, 2017 (ver pp. 10–13).

[12] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (ver p. ii).

[13] L. Monteiro. *Slides da disciplina de teoria da computação*. Rel. téc. FCT-UNL (ver pp. 4, 8, 10, 13, 16, 32).

[14] N. Moreira e R. Reis. *FAdo Documentation Release 2.1.2*. Rel. téc. 2023-04-11 (ver p. 21).

[15] A. Ravara. *Slides da disciplina de teoria da computação*. Rel. téc. FCT-UNL, 2020 (ver pp. 4–7).

[16] *Repositório OCamlFLAT*. URL: https://gitlab.com/releaselab/leaf/OCamlFlat/-/tree/master/src (acedido em 2023-07-11) (ver p. 24).

[17] E. Rich. *Automata, Computability and Complexity: Theory and Applications*. Pearson Education, Inc, 2019 (ver pp. 9, 12, 15, 16).

[18] E. M. F. B. D. Silva. *Gramáticas LL(1) Em OCAML-FLAT/OFLAT*. Novembro, 2021 (ver pp. 24, 29).

[19] B. R. P. D. Sousa. *Análise de sintaxe LR em OCamlFLAT/OFLAT*. Setembro, 2022. URL: https://run.unl.pt/bitstream/10362/151097/1/Sousa_2022.pdf (ver pp. 24, 28).

[20] *what is cytoscape?* URL: https://cytoscape.org/what_is_cytoscape.html (acedido em 2023-07-11) (ver p. 29).

[21] Wikipedia. *Context-sensitive grammar*. 2023-05-25. URL: https://en.wikipedia.org/wiki/Context-sensitive_grammar (acedido em 2023-07-04) (ver p. 15).

[22] Wikipedia. *Recursively enumerable language*. 2023-05-18. URL: https://en.wikipedia.org/wiki/Recursively_enumerable_language (acedido em 2023-07-04) (ver p. 15).