UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Diogo João Cardoso

ASYNCRFJ: UMA ABORDAGEM ASSÍNCRONA À PROGRAMAÇÃO ORIENTADA A OBJETO REATIVA

Santa Maria, RS
2018

Diogo João Cardoso

ASYNCRFJ: UMA ABORDAGEM ASSÍNCRONA À PROGRAMAÇÃO ORIENTADA A OBJETO REATIVA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

ORIENTADORA: Prof.ª Juliana Kaizer Vizzotto

Santa Maria, RS
2018

Cardoso, Diogo João
AsyncRFJ: Uma Abordagem Assíncrona à Programação Orientada a Objeto Reativa / Diogo João Cardoso.- 2018.
92 p.; 30 cm

Orientadora: Juliana Kaizer Vizzotto
Dissertação (mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Programa de Pós-Graduação em Informática, RS, 2018

1. Programação Reativa 2. Programação Orientada a Objetos 3. Formalização de Linguagens 4. Interpretador I. Kaizer Vizzotto, Juliana II. Título.

Sistema de geração automática de ficha catalográfica da UFSM. Dados fornecidos pelo autor(a). Sob supervisão da Direção da Divisão de Processos Técnicos da Biblioteca Central. Bibliotecária responsável Paula Schoenfeldt Patta CRB 10/1728.

End. Eletr.: dcardoso@inf.ufsm.com

Diogo João Cardoso

ASYNCRFJ: UMA ABORDAGEM ASSÍNCRONA À PROGRAMAÇÃO ORIENTADA A OBJETO REATIVA

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação.

Aprovado em 13 de julho de 2018:

Juliana Kaizer Vizzotto, Dra. (UFSM)
(Presidenta/Orientadora)

Eduardo Kessler Piveta, Dr. (UFSM)

André Rauber Du Bois, Dr. (UFPel)

Santa Maria, RS
2018

RESUMO

ASYNCRFJ: UMA ABORDAGEM ASSÍNCRONA À PROGRAMAÇÃO ORIENTADA A OBJETO REATIVA

AUTOR: Diogo João Cardoso
ORIENTADORA: Juliana Kaizer Vizzotto

A utilização de programação reativa tem se tornado cada vez mais comum em sistemas atuais, se destacando principalmente por conseguir manipular fluxos de dados de entrada. No paradigma de programação reativa, um novo valor no fluxo de dado pode ser visto como um evento, e um evento pode causar mudanças no programa, que devem ser devidamente propagadas. Nesta dissertação é discutida a criação de uma linguagem de programação reativa que utiliza-se do paradigma de orientação a objetos (OO), fornecendo a possibilidade de manipular classes e objetos, onde expressões e dados reativos são modelados como uma extensão da proposta Featherweight Java (FJ). Esta extensão é definida formalmente através da apresentação de sua *semântica operacional*, podendo ser implementada em qualquer linguagem de programação que forneça o mecanismo de *closures*. A formalização desta linguagem permitiu a criação de um interpretador, que implementa as fases de análise léxica, sintática e semântica, com foco especial no tratamento do sistema de tipos para embutir conceitos de computação reativa em uma linguagem clássica, demonstrando a interação entre os dois paradigmas de programação estudados.

Palavras-chave: Programação Reativa. Programação Orientada a Objetos. Programação Funcional. Formalização de Linguagens. Interpretador.

ABSTRACT

ASYNCRFJ: AN ASYNCHRONOUS APPROACH TO REACTIVE OBJECT-ORIENTED PROGRAMMING

AUTHOR: Diogo João Cardoso
ADVISOR: Juliana Kaizer Vizzotto

The presence of reactive programming has been more common in current systems, one of the highlights of its use is the management of input data streams. In the reactive programming paradigm, a new value for the input stream can be seen as an event, which may cause changes in the program, so it needs to be properly propagated. In this work is discussed the creation of a reactive programming language implementing the object-oriented paradigm (OO), allowing manipulation of classes and objects, where reactive expressions and data are modeled as an extension for Featherweight Java (FJ). This language is formally defined through the *operation semantics*, creating the opportunity to be implemented in any language that provides *closures*. An interpreter for the formalized language is also presented, that includes lexical, syntactic and semantic analysis, with its focus on the type system to embed reactive programming concepts in a classical language, demonstrating the interaction between the two reviewed paradigms.

Keywords: Reactive Programming. Object-Oriented Programming. Functional Programming. Formal Languages. Interpreter.

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 INTRODUÇÃO

Atualmente se tornou comum o uso de sistemas reativos, que são sistemas onde novos valores de entrada podem ser gerados a qualquer momento e sua resposta deve ser imediata, por isso é dito que o programa reage à entrada. Alguns exemplos comuns são, interfaces gráficas de usuário que precisam responder aos comandos durante seu uso, sistemas de *software* embutidos, que precisam reagir aos sinais do *hardware* e controlá-los, e sistemas distribuídos, que precisam reagir aos pedidos vindos da rede.

Além da necessidade de descrever algoritmos para computar uma resposta usando os dados de entrada, semelhante a uma aplicação simples, um sistema reativo também deve reagir às mudanças de entradas e atualizar os dados de saída adequadamente. Dessa forma, o tempo de computação é ainda mais importante, já que sistemas reativos funcionam em tempo-real. Se um comportamento reativo envolve computações não-triviais ou grandes quantidades de dados, várias estratégias de otimização precisam ser aplicadas (como sistema de *cache* e atualizações incrementais).

Valores a serem processados em um sistema reativo podem ser recebidos ao mesmo tempo, ou seja, a assincronia dos dados é uma característica importante do sistema. Como um paradigma para criação de sistemas reativos, programação reativa enfatiza na formulação da lógica de programação assíncrona como um fluxo de dados, e automatiza a propagação de mudanças de valores em variáveis utilizadas no sistema. Linguagens que utilizam tal paradigma de programação provêm funções compostas adequadas, permitindo que a expressividade do processamento de valores reativos se aproxime do processamento de valores comuns da linguagem.

Programação Reativa Funcional (FRP) é uma maneira de programação que combina técnicas da programação funcional e da programação reativa para criar aplicações e serviços. FRP é uma técnica específica para melhorar a escrita de código em uma área que é conhecida por ser fonte de complexidade (e *bugs*): propagação de eventos (BLACKHEATH; JONES, 2016), e pode ser vista como uma linguagem de domínio específico mínima para lógica de estados, tendo uma forte capacidade expressiva.

Conal Elliott, um dos criadores do FRP, descreve o FRP como denotativo. Sendo baseado em uma semântica composicional precisa, simples e independente de implementação, o FRP especifica o significado de cada tipo e bloco. A natureza composicional da semântica, então, determina o significado de todos os tipos corretos e suas combinações e blocos (ELLIOTT, 2011). Assim, um *verdadeiro* sistema FRP deve ser especificado utilizando uma semântica denotacional, que é definida como uma expressão matemática do significado formal de uma linguagem de programação (BLACKHEATH; JONES, 2016). A semântica denotacional provém ao sistema FRP uma especificação formal do sistema e provas de importantes propriedades.

O trabalho de Czaplicki e Chong (2013) implementa a linguagem Elm, uma linguagem de programação reativa funcional que tem como objetivo trabalhar com interfaces gráficas para usuários de forma assíncrona. É demonstrado, a partir de uma semântica operacional e sistema de tipos, que a linguagem não é ambígua, ou seja, sua definição atribui todas as expressões permitidas pela linguagem à interpretações ou valores únicos.

O Featherweight Java (FJ) é uma proposta *core* mínima para a linguagem Java, proposto por Igarashi, Pierce e Wadler (2001). Uma das principais características do FJ é escolha de ser compacto. O objetivo é omitir o máximo de características possíveis do Java, e ao mesmo tempo possibilitar a modelagem do sistema de tipos. Porém, esse fragmento é amplo o suficiente para incluir vários programas, com o objetivo de prover uma maneira fácil de aplicar provas e estudar as consequências de extensões e variações em regras semânticas.

Neste trabalho, apresenta-se *AsyncRFJ*, uma linguagem orientada a objeto reativa com suporte à assincronia, projetada para prover um mecanismo para escrita de programas utilizando construções reativas em um pequeno *subset* de uma linguagem orientada a objeto amplamente utilizada. Sua semântica formal estende Featherweight Java, um pequeno *calculus* com uma definição semântica específica para os principais aspectos do Java. A linguagem utiliza um conjunto de construtores reativos baseados no Elm, uma linguagem FRP focada especificamente para interfaces gráficas para usuários. Tecnicamente, a principal contribuição do trabalho é apresentar, de uma maneira simples, como descrever formalmente a semântica de programação reativa assíncrona no contexto de uma linguagem orientada a objetivos. Também, disponibilizar um interpretador e *typechecker* para auxiliar o entendimento e utilização da linguagem a partir de exemplos funcionais.

Este trabalho está dividido da seguinte forma: o Capítulo 2 apresenta os principais conceitos referentes à programação reativa funcional, e uma breve descrição da linguagem Elm, incluindo a definição e manipulação de seus construtores reativos. O Capítulo 3 descreve a linguagem FJ, a proposta original de sua sintaxe e semântica, bem como o funcionamento de seu sistema de tipos. O Capítulo 4 descreve a linguagem proposta deste trabalho, incluindo as extensões feitas na linguagem FJ para possibilitar a utilização de construtores reativos, bem como sua verificação de tipo e execução de um programa na linguagem. Já o Capítulo 5 apresenta um interpretador implementado para a linguagem definida neste trabalho, utilizando a linguagem *Haskell*. O Capítulo 6 apresenta trabalhos e estudos relacionados a este trabalho. Por fim, o Capítulo 7 apresenta as conclusões e possíveis trabalhos a serem desenvolvidos no futuro.

2 PROGRAMAÇÃO REATIVA FUNCIONAL

Programação Reativa (*Reactive Programming*) é um paradigma em que novos dados podem ser recebidos e processados fora de ordem, diferente da maneira convencional, onde os dados são normalmente processados sequencialmente. Outra característica do paradigma é que esses dados também podem ser processados ao mesmo tempo, esse tipo de processamento é nomeado assíncrono, que é uma das principais características de um programa reativo.

A definição de Staltz (2014) diz que programação reativa trabalha com fluxos de dados de maneira assíncrona. Um fluxo de dado, também chamado de *stream*, é uma maneira de representar geração de dados em relação ao tempo, e não é necessariamente constante, levar em consideração o tempo de espera entre um dado e outro é uma escolha específica para cada programa reativo. Exemplos mais comuns de *streams* de dados são, conteúdos de uma interface de usuário como *mouse* ou teclado, ou até mesmo requisições a um servidor recebidas pela rede.

Programação Reativa Funcional (*Functional Reactive Programming, FRP*) é uma maneira de programação que combina técnicas da programação funcional e da programação reativa para criar aplicações e serviços. O paradigma permite a mudança de estado ou operação de sua plataforma base dinamicamente, com a utilização de eventos e procedimentos. A definição de um evento é, o recebimento de um novo valor para uma *stream*, e um procedimento (adaptado do inglês, *behavior*) é um conjunto de operações realizadas em cima um evento. Eventos e *behaviors* podem ser gerados de maneira assíncrona, o que torna FRP uma ótima solução para desenvolver aplicações com esse tipo de dados (BAINOMUGISHA et al., 2013).

A implementação de FRP normalmente vem em forma de uma biblioteca simples na linguagem sendo utilizada, e substitui os *listeners* (também conhecidos como *callbacks*) no padrão *Observer*. Isso resulta em um código mais limpo, robusto e mais fácil de realizar manutenção, ou seja, simples (BLACKHEATH; JONES, 2016). FRP introduz novas funcionalidades para o progarmador, fazendo com que o código sofra algumas transformações que melhoram seu raciocínio. Mesmo assim, sua compatibilidade com a maneira normal de escrever código é alta, se tornando fácil de incorporar em projetos já existentes (BLACKHEATH; JONES, 2016).

A introdução do FRP foi feita por Elliot e Hudak (ELLIOTT; HUDAK, 1997). Abordagens posteriores refinaram e estenderam o conceito, focando principalmente na semântica formal de tempo contínuo (NILSSON; COURTNEY; PETERSON, 2002). Atualmente, FRP tem sido aplicado a diversos cenários práticos com sucesso, que inclui coordenação de robôs (HUDAK et al., 2002), programação de *network switches* (FOSTER et al., 2011) e sensores de redes *wireless* (NEWTON; MORRISETT; WELSH, 2007). Trabalhos recentes

focam na otimização e garantia de segurança fornecida pelo sistema de tipos. Por exemplo, Krishnaswami, Benton e Hoffmann (2012) usam tipos lineares para controlar alocação de novos nós no grafo de dependência e evitar *memory leaks*.

A ideia de composicionalidade, como uma propriedade matemática, vem de linguística e semântica. Segundo Westerståhl (1998), composicionalidade é a propriedade em que o sentido de uma expressão é determinado pelo sentido de suas partes e as regras usadas para combiná-las. De acordo com Blackheath e Jones (2016), FRP é composicional porquê impõe regras matemáticas que forçam a interação dos elementos de um programa a serem simples e previsíveis. Dessa maneira, a complexidade dos elementos de um programa não fogem do controle rapidamente, fazendo com que a complexidade geral seja proporcional ao tamanho do programa. FRP faz isso em um nível bom de granularidade, fazendo com que a refatoração seja uma tarefa fácil, pois seus fragmentos de código são composicionais, e essa característica permeia pelo código inteiro.

Algo importante que programação funcional provém é o uso de abstrações, elas são consideradas baratas e isso faz com que sejam desejáveis. FRP não é exceção, com abstrações é possível esconder detalhes e simplificar interações, e na prática isso é realizado por técnicas simples que limitam o escopo. Abstrações adequadas conseguem simplificar o código fonte e podem torná-lo mais simples, porém existe a chance de ser necessário tempo extra para entender o significado de dada abstração (BLACKHEATH; JONES, 2016).

2.1 ELM

Elm (CZAPLICKI; CHONG, 2013) é uma linguagem de programação reativa funcional que visa simplificar a criação de interfaces gráficas para usuários (*Graphical User Interfaces*, GUIs) responsivas, e especificamente GUIs para aplicações web. O FRP define variáveis que variam no tempo como *sinais*, e faz o tratamento dessas variáveis utilizando uma metodologia baseda no paradigma de programação funcional. FRP é uma abordagem promissora para implementação de GUIs, onde os sinais podem representar entrada e saída (incluindo interação com usuário, pedidos e respostas do servidor), e outras informações sobre o ambiente de execução. Através da aplicação de programação puramente funcional, a linguagem Elm provém aos programadores a habilidade de, explicitamente, fazer a modelagem de relações complexas dependentes do tempo em uma maneira declarativa de alto nível, utilizando construtores que aplicam funções anônimas à variáveis sinais específicas.

A semântica da maioria das linguagens que suportam FRP assumem que os sinais mudam constantemente. Assim, a implementação processa os sinais o mais rápido possível, e refaz a computação do programa constantemente com os novos valores dos si-

nais. Contudo, na prática muitos sinais mudam de maneira discreta e não frequente, assim o processamento constante de sinais gera computação desnecessária. Em contrapartida, Elm assume que todos os sinais são discretos, e usa essa suposição para detectar quando um sinal não sofreu mudanças, e neste caso, evitar computações desnecessárias.

Um sinal muda somente quando um evento discreto ocorre, e um evento ocorre quando a entrada de programa (posição do mouse, por exemplo) muda. O resultado de um programa é gerado a partir do processamento de suas variáveis, se um evento gerar um novo valor para alguma dessas variáveis, o programa deve ser recomputado. Implementações anteriores de FRP exigem que os eventos sejam processados de maneira síncrona: um de cada vez na exata ordem de ocorrência. Pois a sincronização é necessária para permitir que o programador raciocine sobre o comportamento do sistema FRP, e garantir sua corretude e funcionalidade.

Assim como trabalhos anteriores em FRP, Elm restringe o uso de sinais para permitir uma implementação eficiente. A linguagem leva em consideração a implementação de Wan, Taha e Hudak (2002), onde sinais discretos generalizam sinais contínuos, e uma versão discreta de *Arrowized* FRP (NILSSON; COURTNEY; PETERSON, 2002). Além disso, é possível que o programador especifique quando a atualização de um sinal deve ser computada de forma assíncrona, aumentando o poder de expressão da linguagem.

O exemplo 2.1 demonstra a capacidade do Elm de executar computações assíncronas com sinais. O programa usa palavras inseridas pelo usuário para realizar um *fetch* em uma imagem de um serviço web (por exemplo Flickr), e essa operação pode demorar para ser executada. Ao mesmo tempo, é mostrada a posição atual do mouse (valor é atualizado em um intervalo de tempo), independente do tempo que demora para fazer a busca da imagem.

Código-fonte 2.1 – Exemplo de código em Elm

```
1  (inputField, tags) = Input.text ``Enter a tag''
2
3  getImage tags =
4      lift (fittedImage 300 300)
5           (syncGet (lift requestTag tags))
6
7  scene input pos img =
8      flow down [ input, asText pos, img ]
9
10 main = lift3 scene inputField
11                   Mouse.position
12                   (async (getImage tags))
```

Fonte: Produzido pelo autor.

O código `Input.text ``Enter a tag''` cria um campo entrada para texto e retorna um par `(inputField, tags)`, onde `inputField` é um sinal para elementos gráficos que

serão utilizados, e `tags` é um sinal para *strings*. Toda vez que o texto do campo de entrada muda (devido ao usuário digitando um novo valor), os dois sinais produzem um novo resultado: o elemento gráfico atualizado, e o texto inserido no campo de entrada.

A função `getImage` recebe um sinal de *strings*, e retorna um sinal de imagens. Para cada *string* que o usuário insere, essa função realiza um pedido a um servidor, recebendo uma URL de uma imagem com as mesmas *tags* (i.e., palavras-chave) da *string* inserida. A função `requestTag` recebe uma *string* e a transforma em um pedido HTTP apropriado para o servidor. A expressão `lift requestTag tags` retorna um sinal desses pedidos HTTP. A função embutida `syncGet` realiza os pedidos e avalia para um sinal de URLs de imagens. Por fim, a função `fittedImage` recebe dimensões e URL de imagem vinda dos pedidos, e constrói o elemento da imagem.

O tipo de variável *Element* indica um elemento gráfico, ou um elemento composto por elementos gráficos, que pode ser exibido. A função `scene` recebe os argumentos `input` (um *Element*), `pos` (um valor arbritrário) e `img` (um *Element*), e os constrói verticalmente.

Já a chamada de `main` junta todas as peças, aplicando a função `scene` a três sinais: o sinal de elementos representando o campo de entrada em que o usuário digita, a posição do mouse (`Mouse.position`), e o sinal de imagens vindas da entrada do usuário (`getImage tags`). A função `lift3` reaplica a função `scene` aos três sinais, toda vez que qualquer um deles produz um novo valor.

A palavra-chave `async` indica que a computação de `getImage` deve acontecer de forma assíncrona. Sem o uso da palava-chave *async*, o programa deve respeitar a ordem global dos eventos: quando o sinal de tags gerar uma nova *string* , o processamento do novo valor de `Mouse.position` deve esperar a busca da nova imagem no servidor.

O construtor *async* pode ser aplicado a qualquer sinal, e fornece uma maneira simples para o programador especificar quando a computação sobre um sinal deve ocorrer de maneira assíncrona. Isso permite que computações demoradas não resultem em uma GUI não responsiva.

2.1.1 Sintaxe

Para a definição formal da linguagem Elm, Czaplicki e Chong (2013) definem a linguagem FElm (Featherweight Elm), e sua sintaxe é apresentada na Figura 2.1. Expressões da linguagem incluem: o valor unidade (); inteiros n; variáveis x; funções $\lambda x : \eta.e$; aplicações $e_1\ e_2$; expressões `let` $x = e_1$ `in` e_2; e expressões condicionais `if` $e_1\ e_2\ e_3$; formando todas as construções base para linguagens funcionais. A expressão condicional `if` $e_1\ e_2\ e_3$ avalia para e_2 se e_1 avaliar para um valor diferente de zero, caso contrário a expressão será avaliada para e_3. Por fim, a metavariável $\oplus$ representa as operações binárias possíveis com valores inteiros.

Figura 2.1 – Definições Sintáticas do FElm.

$$
\begin{array}{rl}
\textit{Números} & n \in \mathbb{Z} \\
\textit{Variáveis} & x \in \text{Var} \\
\textit{Sinais de entrada} & i \in \text{Input} \\
\textit{Expressões} & e ::= () \mid n \mid x \mid \lambda x : \eta . e \mid e_1\ e_2 \mid e_1 \oplus e_2 \\
 & \mid \text{if } e_1\ e_2\ e_3 \mid \text{let } x = e_1 \text{ in } e_2 \mid i \\
 & \mid \text{lift}_n\ e\ e_1 \ldots e_n \mid \text{foldp } e_1\ e_2\ e_3 \\
 & \mid \text{async } e \\
\textit{Tipos Simples} & \tau ::= \text{unit} \mid \text{int} \mid \tau \to \tau' \\
\textit{Tipos de Sinais} & \sigma ::= \text{signal } \tau \mid \tau \to \sigma \mid \sigma \to \sigma' \\
\textit{Tipos} & \eta ::= \tau \mid \sigma
\end{array}
$$

Fonte: Produzido pelo autor.

É definido um conjunto de identificadores chamado `Input` para sinais de entrada recebidos de ambientes externos, esse cojunto de identificadores é representado por *i*. Sinais podem ser vistos como uma *stream* de valores. Por exemplo, um sinal de entrada representando a largura de uma janela pode ser visto como uma *stream* de valores inteiros, e um novo valor para o sinal é gerado toda vez que a largura da janela muda.

Um *evento* ocorre toda vez que um sinal de entrada gera um novo valor. Eventos podem iniciar computação. No FElm, todo sinal sempre tem um valor atual: o valor gerado mais recentemente, ou para um sinal que ainda não gerou um valor, um valor padrão mais apropriado para o sinal. Portanto, todo sinal de entrada deve ter um valor padrão, que induz valores padrão para outros sinais.

2.1.2 Manipulação de sinais

As expressões *lift*$_n$*, foldp* e *async* são primitivas para manipular sinais, e serão descritas a seguir. A primitiva, *lift*$_n$, permite que o programador realize transformações e combinações de sinais. Por exemplo, para um número natural $n \geq 0$, a primitiva $lift_n\ e\ e_1 \ldots e_n$ gera a possibilidade da função *e* transformar e combinar os sinais $e_1 \ldots e_n$.

Assim, se *e* é uma função de tipo $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$, e as expressões e_i são de tipo τ_i, então $lift_n\ e\ e_1 \ldots e_n$ aplica a função e aos valores dos sinais $e_1 \ldots e_n$, e a expressão *lift* inteira é do tipo $signal\ \tau$ (i.e., é um sinal que produz valores do tipo τ). A Figura 2.2 apresenta um exemplo de um sinal de inteiros, onde o valor final é obtido pelo cálculo do dobro do valor recebido de um outro sinal *Window.width*.

Figura 2.2 – Exemplo da função *lift*.

$$\text{lift}_1\ (\lambda \text{x:int. x+x})\ \text{Window.width}$$

Fonte: Produzido pelo autor.

A primitiva *lift* apenas opera com os valores atuais de um sinal. Em contrapartida, a primitiva *foldp* trabalha com transformações dependente do passado, podendo computar o valor atual e anteriores de um sinal. A primitiva aplica um *fold* em um sinal do passado, de uma maneira similar que a função padrão *foldl* aplica um *fold* em listas à esquerda da lista.

O tipo de *foldp* é $(\tau \to \tau' \to \tau') \to \tau' \to signal\ \tau \to signal\ \tau'$ para qualquer τ e τ'. Considera-se a expressão $foldp\ e_1\ e_2\ e_3$, onde e_1 é uma função de tipo $(\tau \to \tau' \to \tau')$, e_2 é um valor de tipo τ' e e_3 é um sinal que produz valores de tipo τ. O tipo τ' é o tipo do acumulador, e e_2 é seu valor inicial. Conforme valores são gerados pelo sinal e_3, a função e_1 é aplicada a esses novos valores e o acumulador atual, para produzir um novo valor que será guardado no acumulador. A expressão $foldp\ e_1\ e_2\ e_3$ avalia para um sinal de valores do acumulador.

Por exemplo, supondo que *Keyboard.lastPressed* é um sinal de entrada que indica a última tecla apertada no teclado. O sinal apresentado na Figura 2.3 faz a contagem do número de teclas apertadas (assume-se que cada tecla apertada é processada como um valor inteiro *k*).

Figura 2.3 – Exemplo da função *foldp*.

foldp (λk:int. λc:int. c+1) 0 Keyboard.lastPressed

Fonte: Produzido pelo autor.

Já para composições assíncronas, a primitiva *async* permite que o programador especifique quando a computação em um sinal pode ser feita de maneira assíncrona sem gerar problemas para o resto do programa. Essa anotação permite que seja especificado quando é permitido ignorar a ordem gloval de eventos. O uso e benefício desse construtor é descrito de maneira mais detalhada na seção de semântica do FElm.

2.1.3 Sistema de Tipos

O sistema de tipo da linguagem FElm contém dois tipos: *tipos simples* τ e *tipos de sinais* σ, apresentados na Figura 2.1. Os tipos simples incluem os tipos básicos (valor *unit* e inteiros) e funções de tipos simples ($\tau \to \tau'$). Já os tipos de sinais σ incluem o tipo *signal* τ (para sinais que produzem valores do tipo τ), e funções que produzem tipos sinais ($\tau \to \sigma$ e $\sigma \to \sigma$).

As regras de tipo realizam a exclusão de programas que contém sinais de sinais. A relação de tipo do FElm tem a forma $\Gamma \vdash e : \eta$, indicando que uma expressão e é do tipo η sobre o contexto de tipos Γ, que mapeia variáveis e sinais de entrada com seus tipos. A Figura 2.4 apresenta as regras de inferência para essa relação.

As regras para os operadores de primitivas que manipulam sinais também são descritas na Figura 2.4, em conjunto com as restantes regras da linguagem. Vale notar que a

Figura 2.4 – Regras de inferência de tipo $\Gamma \vdash e : \eta$.

$$\textit{T-Unit}\ \frac{}{\Gamma \vdash () : \text{unit}} \qquad \textit{T-Number}\ \frac{}{\Gamma \vdash n : \text{int}} \qquad \textit{T-Var}\ \frac{\Gamma(x) = \eta}{\Gamma \vdash x : \eta} \qquad \textit{T-Input}\ \frac{\Gamma(i) = \text{signal } \tau}{\Gamma \vdash i : \text{signal } \tau}$$

$$\textit{T-Lam}\ \frac{\Gamma, x : \eta \vdash e : \eta'}{\Gamma \vdash \lambda x : \eta.\ e : \eta \to \eta'} \qquad \textit{T-Async}\ \frac{\Gamma \vdash e : \text{signal } \tau}{\Gamma \vdash async\ e : \text{signal } \tau}$$

$$\textit{T-OP}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \oplus e_2 : \text{int}} \qquad \textit{T-App}\ \frac{\Gamma \vdash e_1 : \eta \to \eta' \quad \Gamma \vdash e_2 : \eta}{\Gamma \vdash e_1\ e_2 : \eta'}$$

$$\textit{T-Cond}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \eta \quad \Gamma \vdash e_3 : \eta}{\Gamma \vdash \text{if } e_1\ e_2\ e_3 : \eta} \qquad \textit{T-Let}\ \frac{\Gamma \vdash e_1 : \eta \quad \Gamma, x : \eta \vdash e_2 : \eta'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \eta'}$$

$$\textit{T-Lift}\ \frac{\Gamma \vdash e : \tau_1 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash e_i : \text{signal } \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \text{lift}_n\ e\ e_1 \ldots e_n : \text{signal } \tau}$$

$$\textit{T-Fold}\ \frac{\Gamma \vdash e_f : \tau \to \tau' \to \tau' \quad \Gamma \vdash e_b : \tau' \quad \Gamma \vdash e_s : \text{signal } \tau}{\Gamma \vdash \text{foldp } e_f\ e_b\ e_s : \text{signal } \tau'}$$

Fonte: Produzido pelo autor.

regra de tipo para expressões de condição *if* $e_1\ e_2\ e_3$ exige que expressão e_1 seja do tipo inteiro, e em particular, não pode ser um sinal. Um programa é *bem tipado* se $\Gamma \vdash e : \eta$ garante que um ambiente de tipo Γ_{input} faz o mapeamento cada entrada $i \in Input$ para um tipo sinal σ.

2.1.4 Semântica

Um programa na linguagem FElm é avaliado em duas etapas. Na primeira etapa, todos os construtores funcionais são avaliados, resultando em um termo em uma linguagem intermediária que mostra claramente como os sinais são conectados. Essa linguagem intermediária é semelhante à linguagem fonte do *Real-Time FRP* (WAN; TAHA; HUDAK, 2001) e *Event-Driven FRP* (WAN; TAHA; HUDAK, 2002). Já na segunda etapa, os sinais são avaliados: a chegada de novos valores nos sinais de entrada produz computações em

uma maneira *push*.

2.1.4.1 Avaliação funcional

A linguagem intermediária é definida de acordo com a gramática apresentada na Figura 2.5. Nesta linguagem intermediária é feita a separação de valores de tipos simples (*unit*, números, e funções de tipos simples), e termos de sinais (que serão a base para a avaliação de sinais). Já a Figura 2.6 apresenta a semântica operacional (*small step*) para a avaliação de um programa em FElm. Um programa FElm e avalia para um termo final, que pode ser um valor simples v ou um termo sinal s. As regras de inferência usam o contexto de avaliação E para implementar uma semântica *call-by-value* da esquerda para a direita.

Figura 2.5 – Sintáxe da linguagem intermediária.

$$
\begin{array}{rl}
\textit{Valores} & v ::= () \mid n \mid \lambda x : \tau . e \\
\textit{Termos de sinais} & s ::= x \mid \text{let } x = s \text{ in } u \mid i \mid \text{lift}_n\ v\ s_1 \ldots s_n \\
 & \quad \mid \text{foldp } v_1\ v_2\ s \mid \text{async } s \\
\textit{Termos Finais} & u ::= v \mid s
\end{array}
$$

Fonte: Produzido pelo autor.

Como pode ser visto na Figura 2.6, a regra *Application* converte uma chamada de função para uma expressão *let*. A regra *Reduce* aplica uma beta-redução em expressões $let\ x = v\ in\ e$, mas somente quando x é vinculado a um valor simples. Se x for vinculado a um termo sinal, então (assim como é mostrado pelo contexto $let\ x = s\ in\ E$) a subexpressão e é avalida sem a substituição de x. Isso garante que o termo sinal não seja duplicado e reduz computações desnecessárias na segunda etapa da avaliação, a avaliação de sinais. Essa abordagem é parecida com o *call-by-need calculus*, que evita a duplicação expressões não avaliadas (ARIOLA; FELLEISEN, 1997).

A regra *Expand* amplia o escopo de uma expressão *let* para permitir que a avaliação continue. O contexto F descreve onde a regra *Expand* pode ser aplicada, e inclui todos os contextos onde um valor v simples é necessário. As variáveis livres no contexto F são representadas pela expressão $fv(F[\cdot])$. Assume-se que as expressões são equivalentes até o momento de renomear variáveis vinculadas, e uma variável x sempre pode ser escolhida de maneira que x não pertence à lista de variáveis livres de contexto ($x \notin fv(F[\cdot])$).

Se um programa FElm é bem tipado, então a primeira etapa de avaliação não fica presa, e sempre avalia para um termo final. (Escreve-se $\rightarrow^*$ para a *closure* reflexiva e transitiva da relação $\rightarrow$).

Teorema 1 (*Type Soundness* e Normalização). Têm-se ambiente Γ_{input} que mapeia toda entrada $i \in Input$ para um tipo sinal. Se $\Gamma_{input} \vdash e : \eta$ então $e \rightarrow^* u$ e $\Gamma_{input} \vdash u : \eta$, para um termo final u.

Figura 2.6 – Semântica para avaliação funcional.

$$
\begin{array}{l}
E ::= [\cdot] \mid E\ e \mid v \oplus E \mid \\
\quad \text{if } E\ e_2\ e_3 \mid \text{let } x = E \text{ in } e \mid \text{let } x = s \text{ in } E \mid \\
\quad \text{lift}_n\ E\ e_1\ \ldots\ e_n \mid \text{lift}_n\ v\ s_1\ \ldots\ E\ \ldots\ e_n \mid \\
\quad \text{foldp } E\ e_2\ e_3 \mid \text{foldp } v_1\ E\ e_3 \mid \text{foldp } v_1\ v_2\ E \mid \\
\quad \text{async } E
\end{array}
$$

$$
\begin{array}{l}
F ::= [\cdot]\ e \mid [\cdot] \oplus e \mid v \oplus [\cdot] \mid \text{if } F\ e_2\ e_3 \mid \\
\quad \text{lift}_n\ [\cdot]\ e_1 \ldots e_n \mid \text{foldp } [\cdot]\ e_2\ e_3 \mid \text{foldp } v_1\ [\cdot]\ e_3
\end{array}
$$

Context:

$$\frac{e \to e'}{E[e] \to E[e']}$$

Op:

$$\frac{v = v_1 \oplus v_2}{v_1 \oplus v_2 \to v}$$

Cond-True

$$\frac{v \neq 0}{\text{if } v\ e_2\ e_3 \to e_2}$$

Cond-False

$$\frac{}{\text{if } 0\ e_2\ e_3 \to e_3}$$

Application

$$\frac{}{(\lambda x : \eta.e_1)\ e_2 \to \text{let } x = e_2 \text{ in } e_1}$$

Reduce

$$\frac{}{\text{let } x = v \text{ in } e \to e\ [v/x]}$$

Expand

$$\frac{x \notin fv(F[\cdot])}{F[\text{let } x = s \text{ in } u] \to \text{let } x = s \text{ in } F[u]}$$

Fonte: Produzido pelo autor.

2.1.4.2 Avaliação de Sinais

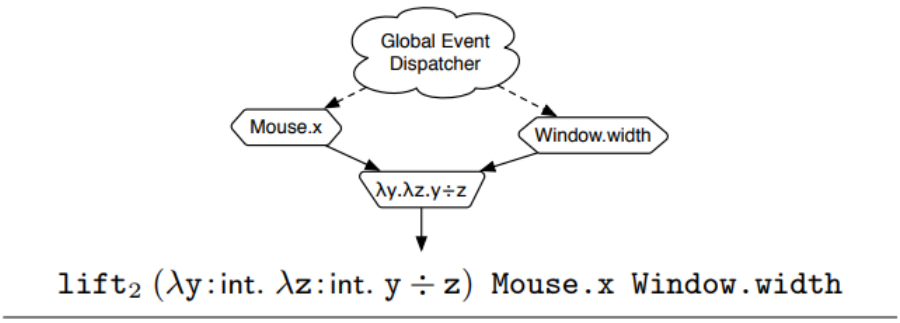
Se um programa FElm avalia para um valor simples v, então o programa não contém sinais e não é um programa reativo. Entretanto, se o programa avalia para um termo sinal s então é feita a segunda de avaliação, que faz a computação ao mesmo tempo que os sinais de entrada produzem novos valores. A seguir será apresentada a técnica usada para avaliação de sinais.

Um termo sinal pode ser representado como um gráfico acíclico, onde os *nós* são sinais de entrada ($e \in$ *Input*), termos *lift*$_n$, e termos `foldp`. A utilização de variáveis é demonstrada nas ligações entre os nós. Essa visualização é chamada de grafo de sinal (*signal graph*).

Nós que representam sinais de entrada não tem ligações (vindas de outros nós), um nó que representa um termo *lift*$_n$ $v\ s_1 \ldots s_n$ tem n ligações de entrada, e um nó que representa um termo *foldp* $v_1\ v_2\ s$ tem uma única ligação de entrada. Grafos de sinais são acíclico já que FElm não consegue representar termos sinais de maneira recursiva. A Figura 2.7 mostra um exemplo de grafo de sinal para um programa FElm simples.

Os nós sem nenhuma ligação de entrada são chamados de *fonte*, esse grupo inclui

Figura 2.7 – Grafo de sinal para a posição x do mouse.



Fonte: Produzido pelo autor.

sinais de entrada e nós *async s*. Esses nós fonte são responsáveis por gerar novos valores, e um evento ocorre quando o sinal representado por um nó fonte produz o novo valor. O elemento responsável por notificar os nós fonte quando um evento ocorre é chamado de *Global Event Dispatcher* (GED). Este *dispatcher* garante que os eventos estão ordenados e não existem dois eventos ocorrendo ao mesmo tempo. No grafo de sinal, uma linha tracejada liga o GED e todos os nós fonte.

Conceituamente, a computação de sinais é síncrona: quando um evento occorre, o novo valor é propagado pelo grafo de sinal inteiro antes que próximo evento seja processado. Contudo, se a semântica utilizar essa abordagem então um atraso global iria existir, pois o processamento de um novo evento estaria bloqueado até todos os eventos anteriores tenham seu processamento finalizado.

Na linguagem Elm, a semântica síncrona é mantida, porém com uma implementação mais eficiente, com o uso de *pipelines* na execução do grafo de sinal: cada nó do grafo tem sua própria *thread* de controle para fazer computações, e cada ligação entre nós utiliza uma fila de valores com o método FIFO (*First In First Out*).

A síncronia da execução pode ser ignorada pelo construtor *async*. O nó para um termo *async s* é um nó fonte: não é ligado a nenhum outro nó de sinal, e recebe notificações do GED. Quando o sinal *s* produz um novo valor, um novo evento é gerado para o termo *async s* e é tratado como um evento vindo do ambiente externo. Este evento é processado pelo *pipeline* no grafo de sinal. O construtor *async*, em combinação com a execução *pipeline*, permite que o programador separe computações de execução longa mantendo a semântica direta de um programa FRP. Assim garantindo que uma GUI se mantenha responsiva mesmo com ações do usuário que geram computações demoradas.

2.2 IMPLEMENTAÇÕES ANTERIORES DE FRP

A versão “tradicional” de Programação Reativa Funcional foi introduzida pelo sistema *Fran* (ELLIOTT; HUDAK, 1997), que por resultado de sua semântica expressiva apresentou problemas severos de eficiência. O sistema permitia que computações dependessem de valores passados, presentes ou futuros. Como resultado, sua implementação tinha que salvar todos os novos valores ocorridos, causando um aumento do uso de memória linear com o tempo.

Real-time FRP (RT-FRP) (WAN; TAHA; HUDAK, 2001) (XU; KHOO, 2002) garante que sinais só possam ser usados usando uma implementação eficiente. Assim como Elm, RT-FRP usa uma linguagem de 2 níveis: uma linguagem base não-restrita (λ-calculus com recursão) e uma linguagem reativa limitada para manipulação de sinais, que suporta recursão porém não suporta funções de alta ordem (tornando-a menos expressiva que o FRP tradicional). RT-FRP garante que as atualizações reativas finalizam desde que os termos da linguagem base finalizem, e o uso de memória não aumenta a não ser que aumente na linguagem base. O que é uma grande melhoria sobre o FRP tradicional.

Event-Driven FRP (E-FRP) (WAN; TAHA; HUDAK, 2002) faz proveito do ganho em eficiência do RT-FRP e introduz a utilização de sinais discretos. Programas em E-FRP são baseados em eventos, no sentido de que nenhuma mudança precisa ser propagada a não ser que um evento tenha ocorrido (i.e., mudança no valor de um sinal) e assim melhora a eficiência da computação.

Arrowized FRP (AFRP) tem como objetivo obter a expressividade do FRP tradicional e manter a eficiência do RT-FRP. Isso é alcaçado exigindo que programadores utilizem uma *função sinal* ao invés de ter acesso direto ao sinal. Uma função sinal é conceituamente equivalente a funções que recebem um sinal de tipo a como entrada, e retornam um sinal de tipo b (LIU; CHENG; HUDAK, 2009)(NILSSON; COURTNEY; PETERSON, 2002).

Problemas semânticos associados a sinais de estado criados dinamicamente são resolvidos criando uma função de sinal para o estado ao invés de um sinal comum. Diferente do FRP tradicional, uma função sinal não pode depender em valores passados arbitrários. Portanto, AFRP permite que programas mudem dinamicamente como sinais são processados sem a ocorrência de *space leaks*. Apesar de não ter acesso direto aos sinais, AFRP alcança a expressividade do FRP tradicional por meio de trocas baseadas em continuação e coleções dinâmicas de funções sinal (SCULTHORPE; NILSSON, 2009)(LIU; HUDAK, 2007)(NILSSON, 2005).

Parallel FRP (PETERSON; TRIFONOV; SERJANTOV, 2000) permite que o processamento de sinais ocorra paralelamente. Os sinais do programa não levam em consideração a ordem de ocorrência dos eventos, permitindo que seu processamento seja fora de ordem. Em um exemplo de servidor, isso significa que pedidos não precisam ser processados na ordem em que são recebidos, então suas respostas podem ser computadas em

paralelo e retornadas imediatamente.

3 FEATHERWEIGHT JAVA

O Featherweight Java (FJ) é uma proposta *core* mínima para a linguagem Java, proposto por Igarashi, Pierce e Wadler (IGARASHI; PIERCE; WADLER, 2001). Uma das principais características do FJ é escolha de ser compacto ao invés de completo. O objetivo é omitir o máximo de características possíveis do Java, e ao mesmo tempo possibilitar a modelagem do sistema de tipos. Porém, esse fragmento é amplo o suficiente para incluir vários programas, com o objetivo de prover uma maneira fácil de aplicar provas e estudar as consequências de extensões e variações em regras semânticas.

É possível utilizar o FJ como ponto de partida para modelar linguagens que estendem o Java. Com a linguagem *core* sendo compacta, pode-se focar nos aspectos essenciais da extensão, motivando seu uso neste trabalho.

3.1 CARACTERÍSTICAS

Featherweight Java tem uma relação com o Java similar à relação entre *Cálculo Lambda* e o *Haskell*. Oferecendo as principais operações da linguagem como classes, métodos, atributos, herança e *casts* dinâmicos com uma semântica muito próxima do Java (IGARASHI; PIERCE; WADLER, 2001).

A definição do FJ contém apenas cinco formas de expressão:

- Criação de Objeto;
- Invocação de métodos;
- Acesso a atributos;
- Conversão de tipos (casts);
- Variáveis;

Assim, os mecanismos do Java que foram omitidos no FJ incluem: atribuição, interfaces, sobrecarga, ponteiros nulos, tipos base (*int*, *bool*, etc.), declaração de métodos abstratos, controle de acesso (*public*, *private*, etc.) e excessões. Características avançadas como concorrência, *inner classes* e *reflection* também são omitidas (PIERCE, 2002). Já as características que são modeladas nessa abordagem incluem, definições de classes mutuamente recursivas, sobrescrita de métodos, recursão de métodos com o uso de *this* , subtipos e *casting* .

Uma das principais simplificações do FJ é a omissão de atribuições. Isso significa que todos os campos e parâmetros de métodos são implicitamente marcados como

final: assume-se que os campos de um objeto são inicializados por seu construtor e após disso permanecem inalterados. Isso restringe FJ a um fragmento “funcional” do Java, em que muitos idiomas comuns da linguagem, como o uso de enumeração, não podem ser representados.

No entando, esse fragmento é computacionalmente completo (possibilitando facilmente a codificação de cálculo *lambda*), e é grande o suficiente para a inclusão de muitos programas úteis (muitos dos programas de Felleisen e Friedman (FELLEISEN; FRIEDMAN, 1998) usam puramente um estilo funcional).

No FJ, um programa consiste em uma coleção de definições de classes e um termo a ser avaliado. Utilizando a proposta de Igarashi, Pierce e Wadler (2001), o seguinte código representa definições de algumas classes:

Código-fonte 3.1 – Exemplo de código em Featherweight Java

```
1  class A extends Object {
2      A() { super(); }
3  }
4  class B extends Object {
5      B() { super(); }
6  }
7  class Pair extends Object {
8      Object fst;
9      Object snd;
10     Pair(Object fst, Object snd) {
11         super();
12         this.fst=fst;
13         this.snd=snd;
14     }
15     Pair setfst(Object newfst) {
16         return new Pair(newfst, this.snd);
17     }
18 }
```

Fonte: Produzido pelo autor.

Para manter uma sintaxe regular, alguns pontos são sempre levados em consideração: sempre incluir uma superclasse (mesmo que seja *Object*); sempre incluir o construtor (até mesmo para classes simples como A e B); e sempre utilizar o *this* em acessos a atributos de uma classe. O construtor segue sempre a mesma forma: um parâmetro para cada atributo, com o mesmo nome desse atributo; e o construtor *super* é sempre invocado para inicializar os atributos da superclasse. O construtor é o único lugar onde aparecem os símbolos *super* e =.

O FJ não suporta o comportamento imperativo, não sendo aplicáveis efeitos colaterais. Assim, não é possível utilizar métodos *set* para modificar o conteúdo de um atributo do objeto. No exemplo acima a função *setfst* realiza a modificação do primeiro atributo do

objeto retornando uma nova instância da classe com esse novo valor, pois a classe *Pair* assim como qualquer outra, é imutável.

No contexto das definições acima, o seguinte termo é avaliado utilizando as regras de substituição que serão apresentadas na seção 3.4.

Código-fonte 3.2 – Exemplo de termo em Featherweight Java

```
1 new Pair(new A(), new B()).setfst(new B())
```

Fonte: Produzido pelo autor.

O resultado da avaliação é o seguinte:

Código-fonte 3.3 – Exemplo de termo após avaliação

```
1 new Pair(new B(), new B())
```

Fonte: Produzido pelo autor.

Este resultado é obtido, mantendo a abordagem funcional, substituindo *new B()* por *newfst* e *new Pair(new A(), new B())* por *this* no corpo do método *newfst* . Depois disso, *this.snd* é substituído por *new B()* no termo *new Pair(newfst, this.snd)*, apresentando como resultado da avaliação *new Pair(new B(), new B())* .

Na sequência serão apresentadas as definições sintáticas, de avaliação semântica e do sistema de tipos da linguagem.

3.2 SINTAXE

A Figura 3.1 apresenta a sintaxe abstrata da linguagem, contendo a gramática *Backus-Naur-Form (BNF)*, onde são apresentadas as declarações de classes (CL), construtores (K), métodos (M), e expressões ou termos (t). Assume-se que o conjunto de variáveis contém a variável especial *this*, que não pode ser utilizada como nome de um argumento para um método. Essa restrição será feita pelas regras de tipo, que serão vistas posteriormente.

Como convenção adotou-se $\overline{\mathtt{f}}$ para designar uma lista de itens $\mathtt{f}_1, \ldots, \mathtt{f}_n$ (similarmente para $\overline{\mathtt{C}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{t}}$, etc.) em todo a linguagem. Além disso, é assumido que sequências de declarações de campos, nomes de parâmetros, e declarações de métodos não são duplicados.

Utilizando essas definições, um programa escrito nesta linguagem pode ser visto como um par (CT, t), sendo uma tabela de classes (CT), e um termo (t). A tabela de classes é uma sequência contendo o nome da classe (C) e a declaração da mesma no formato sintático já apresentado (CL). Como visto, toda classe tem uma superclasse,

Figura 3.1 – Definições Sintáticas do Featherweight Java.

Declarações de classes
CL ::= class C extends C { $\overline{\text{C}}$ $\overline{\text{f}}$; K $\overline{\text{M}}$ }
Declarações de construtores
K ::= C($\overline{\text{C}}$ $\overline{\text{f}}$) { super($\overline{\text{f}}$); this.$\overline{\text{f}}$=$\overline{\text{f}}$; }
Declarações de métodos
M ::= C m($\overline{\text{C}}$ $\overline{\text{f}}$) { return t; }
Termos
t ::= x
| t.f
| t.m($\overline{\text{t}}$)
| new C($\overline{\text{t}}$)
| (C) t

Fonte: Produzido pelo autor.

declarada com a palavra-chave *extends*. No caso especial da classe *Object*, que não tem superclasse, é tratado como um nome de classe reservado e este não aparece na tabela de classes (PIERCE, 2002). Nesta proposta, diferentemente da linguagem Java, os métodos de *Object* são ignorados.

3.3 SISTEMA DE TIPOS

Um dos principais objetivos do Featherweight Java foi a implementação de um sistema de tipos bem definido, tornando a linguagem fortemente tipada. As regras de tipo são focadas na sintaxe, com uma regra para cada forma de expressão (no caso de casts foram definidas 3 regras). Muitas dessas regras são adaptações diretas das regras da linguagem Java.

Primeiro é verificado se a tabela de classes e os termos a serem avaliados estão corretos. A verificação da tabela de classes significa checar se ela está bem definida e com sentido. Em seguida, os termos são checados, verificando se estão declarados de acordo com a tabela de classes, e se as definições de tipos estão corretas. Somente após isto o termo é avaliado.

As linguagens orientadas a objetos trabalham com o mecanismo de herança, e utilizando a tabela de classes é possível constatar a relação de *subtipo* entre classes. A primeira regra da Figura 3.2 demonstra que a relação de *subtipo* é reflexiva: cada classe é *subtipo* de si mesma. Na segunda regra, a relação de *subtipo* é transitiva: se a classe *C* é subtipo da classe *D* e a classe *D* é subtipo da classe *E* , então a classe *C* é subtipo da classe *E*. Já na última regra é possível perceber a relação de herança, a qual é definida na sintaxe da linguagem, através da palavra chave *extends*.

Figura 3.2 – Regras de subtipo para as classes.

$$\frac{}{\text{C} <: \text{C}}$$

$$\frac{\text{C} <: \text{D} \quad \text{D} <: \text{E}}{\text{C} <: \text{E}}$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\ldots\}}{\text{C} <: \text{D}}$$

Fonte: Produzido pelo autor.

Figura 3.3 – Exemplos de funções auxiliares.

Field Look Up:

$$\textit{fields}(\text{Object}) = \bullet$$

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\} \quad \textit{fields}(\text{D}) = \overline{\text{D}}\ \overline{\text{g}}}{\textit{fields}(\text{C}) = \overline{\text{D}}\ \overline{\text{g}},\ \overline{\text{C}}\ \overline{\text{f}}}$$

Method Body Look Up:

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\} \quad \text{B m}(\overline{\text{B}}\ \overline{\text{x}})\{\ \text{return e;}\ \} \in \overline{\text{M}}}{\textit{mbody}(\text{m, C}) = \overline{\text{x}}.\text{e}}$$

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\}\ \text{m} \notin \overline{\text{M}}}{\textit{mbody}(\text{m, C}) = \textit{mbody}(\text{m, D})}$$

Fonte: Produzido pelo autor.

3.3.1 Funções auxiliares

Para as regras de tipagem e regras de redução, é necessário utilizar funções auxiliares. A Figura 3.3 apresenta as definições de alguns exemplos de funções auxiliares (*Field Look Up* e *Method Body Look Up*).

A função *fields* retorna todos os atributos de uma classe e possibilita sua checagem. No caso da classe *Object* não há nenhum atributo para ser retornado (•). Para as demais classes, o retorno da função são os atributos da própria classe e os atributos de sua superclasse, em forma de lista de tuplas que contém o tipo e nome dos atributos ($\overline{\text{D}}\ \overline{\text{g}}$, $\overline{\text{C}}\ \overline{\text{f}}$).

Para descobrir o corpo de um método na classe C, é utilizada a função *mbody*(m, C). É feita a verificação se o método existe na classe atual, caso contrário é chamada a função para sua superclasse. O retorno da função *mbody* é um par que contém a sequência de parâmetros do método e a expressão contida sem seu corpo. A sequência de parâmetros é chamada de $\overline{\text{x}}$ e a expressão é chamada de e, assim retornando o par $\overline{\text{x}}$.e.

Figura 3.4 – Regras para a checagem da tabela de classes.

Regra para classes bem definidas:

$$\frac{K = C(\overline{D}\ \overline{g}, \overline{C}\ \bar{f})\ \{\ super(\overline{g});\ this.\bar{f} = \bar{f}\ \} \qquad \textit{fieds}(D) = \overline{D}\ \overline{g} \qquad \overline{M}\ \text{OK em C}}{\text{class C extends D } \{\ \overline{C}\ \bar{f};\ K\ \overline{M}\ \}\ \text{OK}}$$

Regra para métodos bem definidos:

$$\frac{\overline{x} : \overline{C},\ this : C \vdash t_0 : E_0 \qquad E_0 <: C_0 \qquad \textit{CT}(C) = \text{class C extends D } \{\ \ldots\ \} \qquad \textit{override}(m, D, \overline{C} \rightarrow C_0)}{C_0\ m(\overline{C}\ \overline{x})\ \{\ \text{return } t_0;\ \}\ \text{OK em C}}$$

Fonte: Produzido pelo autor.

3.3.2 Tipagem

O primeiro processo que o sistema de tipos realiza é a verificação se as classes da tabela de classes estão bem declaradas. De acordo com a figura 3.4, em uma classe bem declarada, o construtor deve aplicar *super* aos parâmetros de sua superclasse (caso exista), e inicializar seus atributos internos. Além disso, todos os métodos da classe devem estar corretamente declarados.

Na verificação de classes vale a pena detalhar a utilização da função *fields* , vista anteriormente. Por fim, para verificar se um método está corretamente declarado, é necessário que cada um de seus parâmetros seja de tipos válidos na linguagem, bem como o resultado do processamento do termo (descrito no corpo do método) seja avaliado para o tipo esperado pelo retorno do método. Em caso de *override*, se um método com o mesmo nome está declarado na superclasse, então o tipo deve ser o mesmo.

Após a verificação da tabela de classes, vem a checagem dos tipos presentes no termo a ser avaliado. É introduzida uma estrutura auxiliar chamada *contexto* Γ, que contém um mapeamento de variáveis e seus tipos. Este contexto é preenchido dinamicamente de acordo com as definições presentes nos termos e na tabela de classes.

A primeira regra da figura 3.5 refere-se à tipagem de variáveis, verificando a existência da mesma no contexto, ou seja, a variável *x* é do tipo definido no contexto Γ. O contexto Γ pode ser visto como o *escopo* do bloco sendo avaliado. As regras seguintes apresentam respectivamente os tipos para acesso a atributos, invocação de métodos e criação de objetos.

Em um termo que consiste no acesso a um atributo de uma classe, utiliza-se a regra *T-Field* para tipagem, que leva em consideração o tipo do atributo presente na classe alvo. Primeiro obtêm-se o tipo da expressão e_0, dessa forma é verificado se existe um ar-

Figura 3.5 – Regras de tipos de expressões do Featherweight Java.

T-Var:

$$\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x})$$

T-Field:

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathsf{C}_0 \quad fields(\mathsf{C}_0) = \overline{\mathsf{C}}\ \bar{\mathsf{f}}}{\Gamma \vdash \mathsf{e}_0.\mathsf{f}_i : \mathsf{C}_i}$$

T-Invk:

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathsf{C}_0 \quad mtype(\mathsf{m}, \mathsf{C}_0) = \overline{\mathsf{D}} \to \mathsf{C} \quad \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{e}_0.\mathsf{m}(\overline{\mathsf{e}}) : \mathsf{C}}$$

T-New:

$$\frac{fields(\mathsf{C}) = \overline{\mathsf{D}}\ \bar{\mathsf{f}} \quad \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{C}} \quad \overline{\mathsf{C}} <: \overline{\mathsf{D}}}{\Gamma \vdash \mathsf{new\ C}(\overline{\mathsf{e}}) : \mathsf{C}}$$

Method Type Look Up:

$$\frac{\mathsf{class\ C\ extends\ D\ \{}\overline{\mathsf{C}}\ \bar{\mathsf{f}}\mathsf{;\ K}\ \overline{\mathsf{M}}\mathsf{\}} \quad \mathsf{B\ m(}\overline{\mathsf{B}}\ \overline{\mathsf{x}}\mathsf{)\{\ return\ e;\ \}} \in \overline{\mathsf{M}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{B}} \to \mathsf{B}}$$

$$\frac{\mathsf{class\ C\ extends\ D\ \{}\overline{\mathsf{C}}\ \bar{\mathsf{f}}\mathsf{;\ K}\ \overline{\mathsf{M}}\mathsf{\}\ m} \notin \overline{\mathsf{M}}}{\mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathit{mtype}(\mathsf{m}, \mathsf{D})}$$

Fonte: Produzido pelo autor.

gumento, na classe alvo, com o mesmo nome utilizado no termo avaliado. Se o argumento existir, o tipo do termo de acesso a atributo é o tipo do argumento da classe ($\mathsf{e}_0.\mathsf{f}_i : \mathsf{C}_i$).

Para descobrir o tipo de um método em uma classe C é utilizada a função auxiliar *mtype*(m, C), que retorna um par de valores. A função verifica se o método existe na lista de métodos $\overline{M}$ da classe C, caso contrário é chamada a função *mtype* para sua super classe D. O elemento $\overline{B}$ representa a sequência de tipos dos argumentos do método m, e *B* representa o retorno do mesmo método, assim a função retorna o par $\overline{B} \to B$.

Também é possível obter o tipo de um termo onde é realizada a invocação de um método. A regra *T-Invk* faz a checagem de qual classe o método pertence, para buscar o tipo de seus parâmetros e retorno utilizando a função auxiliar *mtype* vista anteriormente. É feita a verificação se os parâmetros passados na invocação (parâmetros atuais) são de tipos correspondentes aos esperados (parâmetros formais). Caso estes itens estejam em conformidade, é retornado o tipo de retorno do método.

No caso da criação de novos objetos, a regra *T-New* é utilizada. É verificado se os parâmetros passados para o construtor existem na definição da classe alvo, através

da função *fields*, e se seus tipos estão de acordo com os tipos esperados. Caso estejam definidos corretamente, é retornado o tipo (nome da classe) do objeto recém criado.

Figura 3.6 – Regras de tipos para *casts*.

T-UCast:

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathsf{D} \quad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\mathsf{e}_0 : \mathsf{C}}$$

T-DCast:

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathsf{D} \quad \mathsf{C} <: \mathsf{D} \quad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\mathsf{e}_0 : \mathsf{C}}$$

T-SCast:

$$\frac{\Gamma \vdash \mathsf{e}_0 : \mathsf{D} \quad \mathsf{C} \not<: \mathsf{D} \quad \mathsf{D} \not<: \mathsf{C} \quad \textit{stupid warning}}{\Gamma \vdash (\mathsf{C})\mathsf{e}_0 : \mathsf{C}}$$

Fonte: Produzido pelo autor.

Em adição às regras anteriores, exitem 3 regras para *casts* (conversão de tipos), chamadas de *upcast*, *downcast* e a terceira, sendo uma das inovações apresentadas no Featherweight Java, chamada de *stupid cast*. Apresentada na figura 3.6, a regra *T-UCast* mostra que a expressão e_0 tem o tipo D, sendo subtipo do alvo do *cast* (D <: C). A regra *T-DCast* mostra o oposto, onde o tipo alvo é subtipo do termo (C <: D). Já a regra *T-SCast*, mostra que o termo tenta realizar um *cast* onde o tipo do termo e do alvo não são relacionados.

É importante notar que é possível carregar a tabela de classes e usá-la para verificar os tipos de todas as classes antes de avaliar o código fonte do programa. Esta característica similar ao Java permite a manutenção da segurança da linguagem (*type safety*) (PIERCE, 2002).

3.4 AVALIAÇÃO DOS TERMOS

A abordagem do FJ não oferece métodos estáticos e modificadores de acesso, sendo assim, o método *public static void main* utilizado na linguagem Java é representado pelo termo descrito após as definições de classes. Este termo é avaliado após a checagem

Figura 3.7 – Regras de redução.

R-Field:

$$\frac{\textit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}{(\text{new } \mathsf{C}(\overline{\mathsf{v}})).\mathsf{f}_i \rightarrow \mathsf{v}_i}$$

R-Invk:

$$\frac{\textit{mbody}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{x}}.\mathsf{e}_0}{(\text{new } \mathsf{C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{d}}) \rightarrow [\overline{\mathsf{d}} \ / \ \overline{\mathsf{x}}, \text{new } \mathsf{C}(\overline{\mathsf{v}}) \ / \ \text{this}]\mathsf{e}_0}$$

R-Cast:

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})(\text{new } \mathsf{C}(\overline{\mathsf{v}})) \rightarrow \text{new } \mathsf{C}(\overline{\mathsf{v}})}$$

Fonte: Produzido pelo autor.

de tipos e consistência da tabela de classes.

Sendo o último passo na interpretação do programa, nesta etapa o código-fonte está sintaticamente correto, as classe da tabela estão bem definidas, e os tipos dos termos utilizados condizem com as declarações definidas nas classe.

A figura 3.7 apresenta as 3 regras de avaliação (redução) implementadas no FJ: regra para acesso aos atributos de uma classe, para invocação de método e a última para tratamento de *casts*.

A primeira regra, *R-Field*, é usada na avaliação de um termo que faz acesso a um atributo de uma classe, para apresentar seu valor. A regra utiliza a função auxiliar *fields* para obter uma lista de atributos ($\overline{\mathsf{C}}\ \overline{\mathsf{f}}$) da classe utilizada no termo. Após isso busca-se a posição *i* referente ao atributo desejado f_i, e retorna-se o valor respectivo (v_i).

Um termo que faz a chamada de um método é avaliado pela regra *R-Invk*. É feita a busca na tabela de classes do método respectivo à classe informada, através da função auxiliar *mbody* definida na Figura 3.3. A função *mbody* retorna uma lista de parâmetros do método e o termo contido em seu corpo.

Assim, na avaliação de uma chamada de método é feita a substituição dos parâmetros definidos no método ($\overline{\mathsf{x}}$) pelos parâmetros recebidos no termo chamado ($\overline{\mathsf{d}}$), da mesma maneira que as variáveis da classe (*this*) são substituídas pelo novo objeto da classe *C* (new C($\overline{\mathsf{v}}$)) no termo e_0, presente no corpo do método. Essa substituição é representada pela expressão $[\overline{\mathsf{d}} \ / \ \overline{\mathsf{x}}, \text{new } \mathsf{C}(\overline{\mathsf{v}}) \ / \ \text{this}]\mathsf{e}_0$, demonstrando a substituição de $\overline{\mathsf{x}}$ por $\overline{\mathsf{d}}$, e *this* por *new C($\overline{e}$)* no termo e_0.

A última regra trata dos *casts*, *R-Cast*, que faz a verificação se a conversão de tipos é válida no contexto da tabela de classes. Para uma convsersão ser válida, a classe a ser convertida deve ter uma relação subtipo com a classe alvo.

As regras de avaliação definidas acima podem ser aplicadas em qualquer ponto de um termo, portanto há a necessidade de regras de congruência. Essas regras avaliam um termo fazendo a redução para que seja possível que ele se torne um valor, e no final da avaliação este valor representa o fim da computação do termo. A figura 3.8 apresenta a definição das regras de congruência para o FJ.

Figura 3.8 – Regras de congruência.

Congruência para acesso a atributos:

$$\frac{t_0 \to t'_0}{t_0.f \to t'_0.f}$$

Congruência para acesso a métodos:

$$\frac{t_0 \to t'_0}{t_0.m(\bar{t}) \to t'_0.m(\bar{t})}$$

$$\frac{t_i \to t'_i}{v_0.m(\overline{v}, t_i, \bar{t}) \to v_0.m(\overline{v}, t'_i, \bar{t})}$$

Congruência para contrutores:

$$\frac{t_i \to t'_i}{\text{new } C(\overline{v}, t_i, \bar{t}) \to \text{new } C(\overline{v}, t'_i, \bar{t})}$$

Congruência para casts:

$$\frac{t_0 \to t'_0}{(C)\ t_0 \to (C)\ t'_0}$$

Fonte: Produzido pelo autor.

Na proposta original do FJ optou-se por trabalhar com uma relação de redução não determinística, similar a relação de redução *full beta-reduction* do Cálculo Lambda. Cada regra de congruência demonstra a ordem em que os termos são avaliados, e apresentam as definições semânticas que fornecem o comportamento similar à linguagem Java original.

4 A LINGUAGEM ASYNCRFJ

Neste capítulo é descrita a formalização dos recursos adicionados ao Featherweight Java para torná-lo em uma linguagem com suporte a manipulação de dados reativos em uma maneira funcional. Nesta linguagem é utilizado o paradigma de orientação a objeto, seguindo as premissas da proposta original, com a adição de uma camada funcional que permita o processamento de dados reativos. Os principais objetivos a serem alcançados pelo projeto da linguagem são:

- Definir uma semântica operacional para uma extensão ao FJ, que possibilite a utilização de dados reativos.
- Oferecer uma alternativa aos programadores que já utilizam a linguagem Java, mantendo os conceitos do paradigma OO, facilitando a fluência dos futuros usuários da linguagem proposta.
- Possibilitar a definição de operadores reativos na própria linguagem, através da utilização de *closures*, sem necessitar alterações na especificação sintática ou semântica.

As seções seguintes apresentam os recursos da linguagem proposta e exemplos demonstrando sua utilização. No decorrer do capítulo serão definidas as alterações na sintaxe e semântica da linguagem, fornecendo as construções necessárias para apresentar a semântica reativa que trabalha com fluxo de dados. O Apêndice A apresenta a formalização completa da linguagem desenvolvida neste trabalho.

4.1 TIPOS PRIMITIVOS

A definição original do FJ não possui tipos primitivos, portanto é uma das primeiras adições feitas para aprimorar a expressividade de termos reativos. Os tipos tipos primitivos adicionais são *booleanos* e *números inteiros*. Além disso, também há a necessidade de realizar algumas operações sobre estes tipos.

A Figura 4.1 apresenta as construções adicionadas na sintaxe da linguagem. Para tratar dos booleanos, foram adicionadas as palavras-chave que representam os valores *verdadeiro* (*true*) e *falso* (*false*) e também um termo para tratamento de expressões condicionais (*if*). Valores booleanos podem ser modelados como um tipo primitivo da linguagem (PIERCE, 2002). Já para o caso dos números inteiros, foi adicionado uma palavra-chave `int` que representa seu valor. Operadores matemáticos atuam sobre estes números inteiros, tendo a possibilidade de realizar *adição*, *subtração*, *divisão* e *multiplicação* sobre os

Figura 4.1 – Definição sintática para tipos primitivos.

Definição de Tipos

```
T ::= T_FJ | T_P
T_FJ ::= class
T_P ::= boolean | string | int
```

Termos

```
t ::= ···▷ Termos FJ
  | true    | false
  | int
  | string
  | t + t    | t - t
  | t / t    | t * t
  | if ( t ) { t } else { t }
  | let x = t in t
```

Valores

```
v ::= ···▷ Valores FJ
  | true    | false
  | int     | string
```

Fonte: Autoria própria.

mesmos. Além disso, como uma forma de melhorar a leitura e escrita de código, também foi adicionado o operador let, facilitando a manutenção de expressões complexas.

A Figura 4.2 apresenta as regras de avaliação e definição de tipos para as expressões booleanas e condicionais. As duas primeiras regras são axiomas sobre constantes de tipo *booleano*, que avalia para uma das duas possibilidades definidas no código-fonte. Se o primeiro termo for *true* retorna-se t_2, caso contrário retorna-se t_3. Já a terceira regra, representa a congruência do termo condicional, que acontece quando t_1 pode ser avaliado para t'_1, em um passo de redução.

Figura 4.2 – Regras de avaliação e checagem de tipos para booleanos e condicionais.

Regras de avaliação:

$$\frac{}{\text{if (true) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \rightarrow t_2} \qquad \frac{}{\text{if (false) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \rightarrow t_3}$$

$$\frac{t_1 \rightarrow t'_1}{\text{if (} t_1 \text{) \{ } t_2 \text{ \} else \{ } t_3 \text{ \} } \rightarrow \text{if (} t'_1 \text{) \{ } t_2 \text{ \} else \{ } t_3 \text{ \}}$$

Checagem de Tipos:

$$\frac{}{t_1 : \text{boolean}} \qquad \frac{}{t_2 : T} \qquad \frac{}{t_3 : T}$$

$$\frac{t_1 : \text{boolean} \quad t_2 : T \quad t_3 : T}{\text{if (} t_1 \text{) \{ } t_2 \text{ \} else \{ } t_3 \text{ \}}}$$

Fonte: Autoria própria.

As regras de tipo e avaliação para números inteiros e operadores matemáticos são apresentadas na Figura 4.3. As regras de redução para operadores matemáticos demonstram a ordem de avaliação de um termo, da esquerda para a direita, até que todos se tornem valores. As regras de tipo consideram as constantes definidas na sintaxe, e os tipos esperados para operações matemáticas.

Figura 4.3 – Regras de avaliação e checagem de tipos para números inteiros e operadores.

Regras de avaliação:

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \quad \frac{t_2 \rightarrow t_2'}{v_1 + t_2 \rightarrow v_1 + t_2'}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \quad \frac{t_2 \rightarrow t_2'}{v_1 - t_2 \rightarrow v_1 - t_2'}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 * t_2 \rightarrow t_1' * t_2} \quad \frac{t_2 \rightarrow t_2'}{v_1 * t_2 \rightarrow v_1 * t_2'}$$

$$\frac{t_1 \rightarrow t_1'}{t_1 / t_2 \rightarrow t_1' / t_2} \quad \frac{t_2 \rightarrow t_2'}{v_1 / t_2 \rightarrow v_1 / t_2'}$$

Checagem de Tipos:

$$\frac{}{\Gamma \vdash \text{int} : \text{Integer}}$$

$$\frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 + t_2 : \text{int}} \quad \frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 - t_2 : \text{int}} \quad \frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 * t_2 : \text{int}} \quad \frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 / t_2 : \text{int}}$$

Fonte: Autoria própria.

Já a Figura 4.4 apresenta as regras de avaliação e checagem de tipo para o operador *let*. A avaliação indica a substituição do primeiro termo v_1, em sua forma mais reduzida, no termo t_2. Caso não esteja na forma de um valor, o termo será avaliado primeiro. Na checagem de tipo, deve-se adicionar o tipo do termo t_1 no contexto Γ para a avaliação de t_2. O retorno do tipo da expressão `let` inteira, é o tipo do termo t_2.

4.2 CLOSURES

A partir da versão 8 do Java, disponibilizada em 2014, dentre outras funcionalidades, foi feita a introdução de *expressões lambda* ou *closures*. Este novo recurso, herdado do Cálculo Lambda, permite que funções sejam utilizadas como argumentos para métodos, resultando na capacidade de realizar procedimentos complexos com menos linhas de

Figura 4.4 – Regras de avaliação e checagem de tipos para o operador *let*.

Regras de avaliação:

$$\text{let } x = v_1 \text{ in } t_2 \rightarrow [x \mapsto v_1]\ t_2$$

$$\frac{t_1 \rightarrow t_1'}{\text{let } x = t_1 \text{ in } t_2 \rightarrow \text{let } x = t_1' \text{ in } t_2}$$

Checagem de Tipos:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2}$$

Fonte: Autoria própria.

código. Além disso, este nova funcionalidade permite explorar o paralelismo em nível de operações sobre coleções de objetos através da API de *Streams*.

A proposta de Bellia e Occhiuto (2010) tem como objetivo estender o FJ, para permitir a utilização de *closures*. Essa proposta inclui regras sintáticas, semânticas e de tipos que definem o comportamento e garantem a segurança da implementação de *closures* na linguagem. Isso significa que as propriedades de *closures* são modeladas com as características essenciais do FJ. Sendo assim, para trabalhar com *expressões lambda* é preciso estender o sistema de tipos da linguagem, adicionando um tipo que represente uma função, e implementar regras de avaliação para estas funções anônimas. Dessa maneira é possível realizar a passagem de *closures* como parâmetros, permitindo o uso em métodos ou outras expressões.

A implementação real do Java se difere dessa definição, pois não contém o tipo *função*, e *closures* somente podem ser usadas em *interfaces funcionais*. Para simplificar a construção, foi escolhida a modelagem de expressões *lambda* como um tipo, excluindo a necessidade de criar o comportamento completo de interfaces do Java.

Para isso, novas regras sintáticas foram adicionadas na linguagem. A Figura 4.5 apresenta as regras para *expressões lambda*, onde uma expressão consiste em uma lista de parâmetros e um termo que representa o corpo da função anônima. Além disso, a *closure* é definida como um tipo da linguagem (T_F), que é representado pelo mapeamento dos tipos dos parâmetros mais o tipo do retorno da função.

Utilizando as regras sintáticas adicionadas, é possível escrever programas que utilizam *expressões lambda* no termo principal, parâmetros ou corpo de métodos, até mesmo parâmetros de outras *closures*, etc. A Figura 4.6 apresenta as regras para avaliação e checagem de tipo para uma expressão *closure*.

A primeira regra (Figura 4.6) mostra a avaliação de uma função anônima, que substitui os valores recebidos como parâmetros ($\overline{d}$) nos parâmetros formais ($\overline{x}$) que aparecem

Figura 4.5 – Definição sintática para expressões *lambda*.

Definição de Tipos

```
T ::= T_FJ | T_P | T_F
T_F ::= T -> T_FJ | T -> T_P | T -> T_F
```

Termos

```
t ::= ···▷ Termos FJ
  | (T̄ x̄) -> t
  | t.invoke(t̄)
```

Valores

```
v ::= ···▷ Valores FJ
  | (T̄ x̄) -> t
```

Fonte: Autoria própria.

Figura 4.6 – Regras de avaliação e checagem de tipo para *closures*.

Regras de redução:

$$((\overline{T}\ \overline{x}) \to T)\ t.invoke(\overline{d}) \longrightarrow [\overline{d} \mapsto \overline{x}]\ t$$

Regras de Congruência:

$$\frac{t \to t'}{t.invoke(\overline{t}) \to t'.invoke(\overline{t})} \qquad \frac{t \to t'}{t.invoke(\ldots, t_i, \ldots) \to t.invoke(\ldots, t'_i, \ldots)}$$

Checagem de tipo de expressão lambda:

$$\frac{\overline{T}\ OK \quad \Gamma, \overline{x} : \overline{T} \vdash t : T}{\Gamma \vdash (\overline{T}\ \overline{x}) \to t : (\overline{T} \to T)}$$

Checagem de tipo de expressão invoke:

$$\frac{\Gamma \vdash t : (\overline{T} \to T) \quad \Gamma \vdash \overline{t} : \overline{S} \quad \overline{S} <: \overline{T}}{\Gamma \vdash t.invoke(\overline{t}) : T}$$

Fonte: Autoria própria.

no termo t, que representa o corpo da função. Já as regras de congruência, primeiro é avaliado o termo da esquerda da invocação *closure*, e caso ele já seja um valor, é avaliado cada um dos parâmetros da invocação. A avaliação dos parâmetros é feita da esquerda para a direita, mantendo a estratégia *call-by-value* utilizada na linguagem Java.

Na checagem de tipo, a primeira regra faz a verificação se a *closure* está bem definida. Primeiro, o tipo de cada um dos parêmetros formais da função é verificado, checando se estão corretamente definidos de acordo com os tipos primitivos e com as definições na tabela de classes. No caso de estarem todos corretamente definidos, é feita a verificação do tipo do corpo da função, representado pelo termo interno do construtor, e o tipo resultado é a combinação do tipo dos parâmetros e do termo interno.

Para a checagem de tipo de uma expressão *invoke*, primeiro é feita a checagem do termo da esquerda, verificando se é do tipo *função*. Depois é feita a checagem se todos os

parâmetros recebidos pela invocação são subtipos dos parâmetros esperados na definição formal da função. Se todas as verificações estiverem em conformidade, o tipo resultado é o tipo do retorno da função.

4.3 CONSTRUTORES REATIVOS

Programação reativa trabalha com o fluxo de dados, isso faz com que a computação de um programa possa acontecer repetidas vezes, seja o programa inteiro ou seus fragmentos. Dessa maneira, uma linguagem que suporta programação reativa deve diferenciar quais partes do código são denominadas reativas ou não. Isso é feito a partir de construtores que são computados especificamente para tratar de fluxo de dados.

Para este trabalho foram escolhidos os construtores baseados no Elm (CZAPLICKI; CHONG, 2013): `lift`, `foldp`, e `async`, que funcionam como manipuladores de variáveis que recebem valores que mudam com o tempo, chamadas de sinais. Como pode ser visto na Figura 4.7, esses sinais são representados por um novo termo `signal t`, onde `t` representa uma variável que receberá dados reativamente, ou o resultado de uma computação sobre sinais.

Um novo ambiente é necessário para manipular os dados classificados como sinais, chamado de $Input$. Para uma variável normal, sua verificação de tipo e avaliação é utilizado o ambiente Γ, já para o caso de sinais, essas operações utilizam este novo ambiente $Input$. A Figura 4.8 apresenta a regra de tipo para um termo sinal, que caso esteja correto terá o tipo `signal` τ, onde τ representa os tipos básicos da linguagem (T_{FJ} e T_{P}, vistos na Figura 4.1). Toda variável que irá receber um valor vindo de fontes externas (teclado, mouse, requisição *web*, etc) é definida como um sinal e sempre tem um valor inicial, sua semântica será vista posteriormente.

4.3.1 Lift

O construtor `lift` tem a função de transformar e combinar sinais. O termo tem a forma de `lift lam` t_1 `...` t_n, onde *lam* representa uma função anônima, e $t_1 \ldots t_n$ são os sinais que a expressão `lift` tem que reagir. A Figura 4.9 apresenta a regra de tipo para uma expressão `lift`. O primeiro termo, lam, representa a closure que será executada, e seu tipo é verificado utilizando as regras vistas na Figura 4.6. Já os termos $t_1 \ldots t_n$ são os sinais que recebem dados reativamentes, e todos são de tipo `signal` τ, pois são sinais que recebem um dado de tipo τ.

Figura 4.7 – Definição sintática para expressões reativas.

Definição de Tipos
`T ::= T_FJ | T_P | T_F | T_S`
`T_S ::= signal T | T -> T_S`
Termos
`t ::= ···▷ Termos FJ`
`  | signal t`
`  | lift t t_1 ... t_n`
`  | foldp t_1 t_2 t_3`
`  | async t`
Valores
`v ::= ···▷ Valores FJ`
`  | signal v`

Fonte: Autoria própria.

Figura 4.8 – Regras de checagem de tipo para um termo *signal*.

Checagem de tipo:
$$\frac{\mathsf{I}\,(i) = \mathsf{signal}\ \tau}{\mathsf{I} \vdash i : \mathsf{signal}\ \tau}$$

Fonte: Autoria própria.

Figura 4.9 – Regras de checagem de tipo para um termo `lift`.

Checagem de tipo:
$$\frac{\Gamma \vdash lam : \tau_1 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash t_i : \mathsf{signal}\ \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \mathsf{lift}\ lam\ t_1 \ldots t_n : \mathsf{signal}\ \tau}$$

Fonte: Autoria própria.

A execução de um `lift` se baseia na execução da *closure* com os parâmetros recebidos pelos sinais. E toda vez que qualquer um dos sinais de um `lift` gerar um novo valor, a expressão inteira é recalculada.

4.3.2 Foldp

Diferente do `lift`, que somente trabalha com novos valores de uma variável, o construtor `foldp` utiliza o valor resultado de sua computação anterior. Dessa maneira é possível criar expressões mais complexas, um exemplo simples seria um contador de eventos (teclas apertadas, requisições recebidas, contagem de tempo, etc). Um termo `foldp` tem a seguinte forma: `foldp lam a t`, onde *lam* representa a função anônima que

será usada, o termo *a* é o termo chamado de *acumulador* que guarda o resultado da computação anterior, e o termo *t* é o único sinal que receberá valores reativamente.

Figura 4.10 – Regras de checagem de tipo para um termo `foldp`.

Checagem de tipo:

$$\frac{\Gamma \vdash lam : \tau \to \tau' \to \tau' \quad \Gamma \vdash a : \tau' \quad \Gamma \vdash t : \text{signal } \tau}{\Gamma \vdash \text{foldp } lam\ a\ t : \text{signal } \tau'}$$

Fonte: Autoria própria.

Quando uma expressão `foldp` é executada, o valor guardado no acumulador e o valor recebido no sinal *t*, são passados como parâmetros para a função anônima *lam*. Sendo assim, a função anônima deve obrigatóriamente ter dois parâmetros em sua definição formal. Como pode ser visto na Figura 4.10, o acumulador é de tipo τ, o termo sinal *t* é de tipo `signal` τ, e o tipo da expressão `foldp` por completo é `signal` τ, o mesmo tipo do retorno da expressão *lambda*.

4.3.3 Async

O último construtor reativo, `async`, tem como objetivo especificar quando a execução de um trexo do código pode ser feita fora da ordem comum de execução. Casos em que isso pode ser usado geralmente incluem processos com tempo de execuções altos, dessa maneira, é trabalho do programador avaliar quando pode ser favorável que algum processo em específico seja executado fora de ordem. A Figura 4.11 demonstra a checagem de tipo de uma expressão `async`, realizando a checagem de seu termo interno, que deve ser um termo sinal de tipo `signal` τ.

Figura 4.11 – Regras de checagem de tipo para um termo `async`.

Checagem de tipo:

$$\frac{\Gamma \vdash t : \text{signal } \tau}{\Gamma \vdash async\ t : \text{signal } \tau}$$

Fonte: Autoria própria.

A execução de um termo `async` é bem semelhante a um sinal comum, e seu funcionamento completo será visto posteriormente.

Figura 4.12 – Definição de um Programa e *Threads*.

$$
\begin{array}{rl}
\textit{Threads} & \mathsf{T} ::= \mathsf{null} \\
 & \mid t\langle e\rangle : \Gamma \parallel \mathsf{T} \\
\textit{Programa} & \mathsf{P} ::= \mathsf{CT}\ t\ \mathsf{T}
\end{array}
$$

Fonte: Autoria própria.

Figura 4.13 – Linguagem Intermediária para o AsyncRFJ.

$$
\begin{array}{rl}
\textit{Valores} & v ::= () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid \lambda x : \tau.e \mid \mathsf{new\ C}(\overline{v}) \\
\textit{Termos Sinal} & s ::= x \mid \mathsf{let}\ x = s\ \mathsf{in}\ u \mid i \mid \mathsf{lift}\ v\ s_1 \ldots s_n \\
 & \quad \mid \mathsf{foldp}\ v_1\ v_2\ s \mid \mathsf{async}\ s \\
\textit{Termos Finais} & u ::= v \mid s
\end{array}
$$

Fonte: Autoria própria.

4.3.4 Threads

Para possibilitar a execução de expressões de onde a sincronidade é um fator relevante, a noção de concorrência deve ser adicionada à linguagem. Isso é feito com a implementação de *threads* baseadas na definição de Tran e Steffen (2010), porém no caso deste trabalho, *threads* executam somente um termo, e esse termo é sempre reativo (`lift`, `foldp` ou `async`). Vale notar que as *threads* são criadas na avaliação inicial, e somente executam na etapa em que novos valores estão sendo gerados para os sinais dentro dos termos reativos. A Figura 4.12 apresenta a estrutura de um programa na linguagem proposta, que contém uma tabela de classes (contendo suas definições), um termo a ser executado e uma lista de *threads*. Toda *thread* é associada a um ambiente local, que é uma cópia do ambiente em seu tempo de criação. Por fim, o operador $\parallel$ representa a execução em paralelismo de uma ou mais *threads*.

4.4 AVALIAÇÃO FUNCIONAL

A avaliação de um programa em *AsyncRFJ* é feita em duas etapas, primeiro é feita a avaliação funcional e depois a avaliação de sinais. A primeira etapa consiste na redução do termo do programa utilizando as definições da tabela de classe, e o resultado é uma lista de *threads* contendo termos intermediários (que contém variáveis sinais). Esses termos intermediários são a ligação entre as duas etapas de avaliação, e são apresentados na Figura 4.13. Vale notar, que se o termo inicial do programa é avaliado para um *valor* desta linguagem intermediária, significa que ele não é um termo reativo e avaliações posteriores não são necessárias.

Na etapa de avaliação funcional, a redução de sinais é feita utilizando seu valor

Figura 4.14 – Contextos de avaliação para o AsyncRFJ.

$$
\begin{aligned}
E ::= {} & [\cdot] \mid E\ e \mid v \oplus E \mid \\
& \text{if } E\ e_2\ e_3 \mid \text{let } x = E \text{ in } e \mid \text{let } x = s \text{ in } E \mid \\
& \text{lift}_n\ E\ e_1 \ldots e_n \mid \text{lift}_n\ v\ s_1 \ldots E \ldots e_n \mid \\
& \text{foldp } E\ e_2\ e_3 \mid \text{foldp } v_1\ E\ e_3 \mid \text{foldp } v_1\ v_2\ E \mid \\
& \text{async } E \mid E \text{ . f} \mid E \text{ . m } (e_1 \ldots e_n) \mid v \text{ . m } (v_1 \ldots E \ldots e_n) \mid \\
& \lambda : \eta \text{ . } E \mid E \text{ . invoke}(e_1 \ldots e_n) \mid v \text{ . invoke}(v_1 \ldots E \ldots e_n)
\end{aligned}
$$

Fonte: Autoria própria.

Figura 4.15 – Regra de Redução para a expressão `let`.

$$
\frac{x \notin fv(F[\cdot]) \qquad Input' = \ldots, add(x, s)}{\Gamma, Input \vdash E[\text{let } x = s \text{ in } u] \longrightarrow Input' \vdash \text{let } x = s \text{ in } E[u]}
$$

Fonte: Autoria própria.

inicial, e para cada um dos construtores reativos `lift`, `foldp`, e `async`, é criado uma thread que contém o construtor completo, uma cópia do ambiente de variáveis comuns Γ, e uma cópia do ambiente de sinais $Input$. O conteúdo de $Input$ é a lista de sinais de entrada que podem ser utilizados pelo programa, neste ponto da avaliação cada sinal contém somente um valor inicial que será usado para gerar um resultado inicial para o programa inteiro.

A Figura 4.14 demonstra o contexto de avaliação para um programa em *AsyncRFJ*. Durante a avaliação, a expressão *let* é um caso especial, pois é preciso garantir que não ocorra possíveis duplicações de *threads* no caso da variável sendo criada receber um valor de tipo sinal (`let` $x = s$ `in` E). A variável declarada na expressão *let* é inserida no ambiente de sinais $Input$ e é vista como um sinal de entrada comum. Dessa maneira as modificações neste novo sinal são salvas no ambiente de sinais e o restante da expressão a ser avaliada trata a variável como um sinal normal. Assim, não há a necessidade de modificar a sintaxe do programa, tratar variáveis declaradas em expressões *let* de uma maneira diferente, ou até mesmo re-avaliar o termo, evitando a criação de *threads* duplicadas. A regra de avaliação de um *let*, que demonstra a operação no ambiente $Input$, pode ser vista na Figura 4.15.

A avaliação de uma expressão `lift` pode ser vista em três etapas: a criação de sua *thread*, que representa a primeira vez que a expressão é avaliada; a avaliação de seus termos sinais internos; e a execução da expressão em si, aplicando os valores, já finais, na expressão *lambda* definida. As regras para essas três etapas podem ser vistas na Figura 4.16.

Figura 4.16 – Regras de Redução para a expressão `lift`.

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{lift}\ lam\ e_1 \ldots e_n \rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{lift}\ lam\ e_1 \ldots e_n)\ \mathsf{T} \longrightarrow (\mathsf{lift}\ lam\ e_1 \ldots e_n)\ \mathsf{T'}}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{lift}\ lam\ s_1 \ldots e \ldots e_n)\ \mathsf{T} \longrightarrow (\mathsf{lift}\ lam\ s_1 \ldots e' \ldots e_n)\ \mathsf{T'}}$$

$$\frac{Input(s_i) = v_i\ \forall i \in (1..n)}{\Gamma, Input \vdash \mathsf{lift}\ lam\ s_1 \ldots s_n \longrightarrow lam\ .\ \mathsf{invoke}\ (v_1 \ldots v_n)}$$

Fonte: Autoria própria.

A primeira regra representa o início da avaliação de uma expressão `lift`, nesta etapa é criada a *thread* ($spawn\ t\langle \mathsf{lift}\ lam\ e_1 \ldots e_n \rangle : \Gamma, Input$) que irá re-executar a expressão posteriormente. A nova *thread* criada é então adicionada à lista de *threads* T, gerando uma nova lista T′. O próximo passo, demonstrads pela segunda regra, é avaliar cada sinal que a expressão `lift` irá utilizar: cada um dos n sinais e é avaliado (resultando em um sinal s), e caso algum sinal gere alguma *thread* a partir de sua avaliação, esta nova *thread* também é adicionada à lista T′. Por fim, como pode ser visto na terceira regra da Figura 4.16, avaliação funcional de um `lift` finaliza gerando um valor inicial para a expressão. Como declarado anteriormente, cada sinal obrigatoriamente tem um valor inicial, portanto realizando um *lookup* no ambiente de sinais $Input$, os valores iniciais de cada sinal são aplicados na função anônima *lambda* através de uma expressão *invoke*.

Para o caso de uma expressão `foldp` ou `async`, a avaliação também é feita em três etapas. A figura 4.17 apresenta as regras usadas na avaliação de uma expressão `foldp`. Já a avaliação de uma expressão `async` é demonstrada na Figura 4.18

Inicia-se a avaliação de um termo `foldp` criando a *thread* que corresponde à expressão ($spawn\ t\langle \mathsf{foldp}\ lam\ e_1 \ldots e_n \rangle : \Gamma, Input$) e adicionando-a à lista de *threads* T. Logo após, é feita a avaliação do único sinal e presente na expressão, e caso gere alguma *thread*, é adicionada à lista T′. Por fim, a avaliação da expressão `foldp` gera um valor inicial, aplicando o valor da variável sinal e e do acumulador acc à expressão lambda lam, com a chamada de uma expressão $invoke$, este valor gerado é salvo no acumulador para ser usado nas próximas execuções.

Figura 4.17 – Regras de Redução para a expressão `foldp`.

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{foldp}\ lam\ acc\ e\rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T} \longrightarrow (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T'}}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T} \longrightarrow (\mathsf{foldp}\ lam\ acc\ e')\ \mathsf{T'}}$$

$$\frac{Input(s) = v}{\Gamma, Input \vdash \mathsf{foldp}\ lam\ acc\ s \longrightarrow lam\ .\ \mathsf{invoke}\ (acc\ v)}$$

Fonte: Autoria própria.

Figura 4.18 – Regras de Redução para a expressão `async`.

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{async}\ e\rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{async}\ e)\ \mathsf{T} \longrightarrow (\mathsf{async}\ e)\ \mathsf{T'}}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{async}\ e)\ \mathsf{T} \longrightarrow (\mathsf{async}\ e')\ \mathsf{T'}}$$

$$\frac{Input(s) = v}{\Gamma, Input \vdash \mathsf{async}\ s \longrightarrow v}$$

Fonte: Autoria própria.

Nesta etapa da avaliação, uma expressão `async` não se difere muito de um sinal comum. Primeiro cria-se a *thread*, que irá re-executar a expressão posteriormente, e adiciona-se na lista de *threads* T′. Após isso, é feita a avaliação do termo e e concatena-se suas possíveis *threads* criadas a mesma lista. Por fim, a avaliação do termo e se reduz a um simples *lookup*, que retorna o valor v da variável sinal, utilizando o ambiente de sinais $Input$.

Figura 4.19 – Regra de redução para o *Global Event Dispatcher*.

$$\frac{\forall i \in (0..n)\ \overline{t_i\langle e\rangle}\ g_i \longrightarrow \overline{t_{i+1}\langle e\rangle}\ \overline{\text{Status e}_i}}{\overline{t_0\langle e\rangle}\ g_0, \ldots, g_n \longrightarrow \overline{t_{n+1}\langle e\rangle}\ \overline{\text{Status e}_0}, \ldots, \overline{\text{Status e}_n}}$$

Fonte: Autoria própria.

Figura 4.20 – Definição de um termo `Status`.

$$\begin{aligned} \texttt{Status}\ e\ \texttt{::=}\quad & \texttt{Change}\ e \\ \mid\ & \texttt{NoChange}\ e \end{aligned}$$

Fonte: Autoria própria.

4.5 AVALIAÇÃO DE SINAIS

A segunda etapa da avaliação de um programa *AsyncRFJ* trata dos dados reativos, refazendo o processamento das partes do programa que utilizam esses novos dados, e possivelmente gerando um novo resultado para o programa. Como visto anteriormente, a avaliação funcional resulta em uma linguagem intermediária, e se um programa resulta em valores não-reativos a avaliação do programa finaliza pois a avaliação de sinais não é necessária. Caso contrário, o resultado da avaliação funcional é uma lista de *threads* a serem executadas.

É nesta etapa da avaliação que feita a introdução do *Global Event Dispatcher* (GED), este componente é responsável pela geração de eventos reativos da linguagem. Quando um novo valor é gerado para algum sinal, um evento acontece, e cada evento contém um id representando qual sinal ele está ligado, e um novo valor v. A Figura 4.19 apresenta a regra de avaliação do GED.

A avaliação do GED apresenta o processamento de todos os eventos utilizando a lista de *threads* geradas anteriormente. Cada evento processado é aplicado a todas as *threads*, e cada *thread* faz uma verificação se irá utilizar o novo valor ou não. Para manter a consistência dos dados avaliados, é preciso que cada avaliação de evento (g_n) utilize as *threads* resultantes da avaliação do evento anterior. O principal motivo para isso é a avaliação do construtor `foldp`, que sempre utiliza o resultado de sua avaliação anterior. Dessa maneira, a avaliação do evento g_0 utiliza a lista de threads $\overline{t_0\langle e\rangle}$, gerando uma nova lista chamada de $\overline{t_1\langle e\rangle}$ e uma lista de `Status` que representa o resultado de cada uma das *threads*, definindo se houve mudança ou não utilizando o novo valor vindo do evento. A definição de um termo `Status` pode ser vista na Figura 4.20.

O resultado da avaliação do GED representa o resultado final do programa: a lista final de *threads*, após todas as modificações que os eventos causaram, e uma lista com todos os resultados das *threads* para cada evento avaliado. Essa lista de resultados está na forma de uma lista de listas de `Status`, e pode representar a evolução do programa

Figura 4.21 – Regra de redução para eventos.

$$\frac{Input'(id) = \text{Change } v \quad \overline{Input' \vdash t\langle e\rangle \to t'\langle e\rangle} \quad \overline{\text{getStatus}(t'\langle e\rangle)} = \overline{\text{Status e}}}{\Gamma, Input \vdash \overline{t\langle e\rangle}\ (id, v) \longrightarrow \Gamma, Input' \vdash \overline{t'\langle e\rangle}\ \overline{\text{Status e}}}$$

Fonte: Autoria própria.

Figura 4.22 – Regra de redução para *threads* de termos `lift`.

$$\frac{lam\ \text{OK} \quad \Gamma, Input \vdash e \to s}{\Gamma, Input \vdash t\langle \text{lift } lam\ s_1 \ldots e \ldots e_n\rangle \longrightarrow t\langle \text{lift } lam\ s_1 \ldots s \ldots e_n\rangle}$$

$$\frac{\forall i \in (0..n)\ Input(s_i) = \text{NoChange } ()}{\Gamma, Input \vdash t\langle \text{lift } lam\ s_1 \ldots s_n\rangle \longrightarrow \text{NoChange } ()}$$

$$\frac{\exists i \in (0..n)\ Input(s_i) = \text{Change } e \quad \Gamma, Input \vdash lam.\ \text{invoke } (v_1 \ldots v_n) \to Input' \vdash \text{Change } e'}{\Gamma, Input \vdash t\langle \text{lift } lam\ s_1 \ldots s_n\rangle \longrightarrow Input' \vdash \text{Change } e'}$$

Fonte: Autoria própria.

conforme os eventos ocorreram.

A avaliação geral de eventos é definida pela avaliação do GED, e cada evento em específico é avaliado utilizando a regra definida na Figura 4.21.

Cada avaliação de evento utiliza a lista de *threads* resultante da avaliação do evento anterior e um par de dados (id, v), que representa qual sinal no ambiente $Input$ está sendo modificado e seu novo valor. Este valor é atualizado no ambiente $Input$ com um valor `Status` de `Change` v. Utilizando este novo ambiente de sinais $Input$, todas as *threads* são avaliadas e suas modificações são salvas em uma nova lista $\overline{t'\langle e\rangle}$, e também é obtido o `Status` resultante de cada *thread*. Portanto o resultado da avaliação de cada evento é a lista de *threads* modificadas e seus resultados.

A avaliação de cada *thread* consiste em verificar se o sinal que está sendo atualizado é utilizado pelo termo da *thread*, e fazer sua recomputação caso necessário. Por definição, o termo principal de uma *thread* é sempre um termo reativo (`lift`, `foldp`, ou `async`), sendo assim, as regras de avaliação de uma *thread* são baseadas em seu termo principal. A Figura 4.22 demonstra as regras de avaliação para uma *thread* que contém uma expressão `lift` como termo principal.

A primeira regra da Figura 4.22 demonstra a etapa de redução dos termos sinais da expressão `lift`. Vale notar que neste ponto da avaliação, o ambiente de sinais $Input$ contém somente uma variável cujo valor atribuído é `Change` e. Sendo assim, a avaliação da expressão `lift` é definida a partir dos dois casos possíveis: se utiliza a variável modi-

Figura 4.23 – Regra de redução para *threads* de termos `foldp`.

$$\frac{lam \text{ OK} \quad \Gamma, Input \vdash e \to s}{\Gamma, Input \vdash t\langle \text{foldp } lam\ acc\ e\rangle \longrightarrow t\langle \text{foldp } lam\ acc\ s\rangle}$$

$$\frac{Input(s) = \text{NoChange } ()}{\Gamma, Input \vdash t\langle \text{foldp } lam\ acc\ s\rangle \longrightarrow \text{NoChange } ()}$$

$$\frac{Input(s) = \text{Change } e \quad \Gamma, Input \vdash lam.\ \text{invoke } (acc\ v) \to Input' \vdash \text{Change } e'}{\Gamma, Input \vdash t\langle \text{foldp } lam\ acc\ s\rangle \longrightarrow Input' \vdash \text{Change } e'}$$

Fonte: Autoria própria.

ficada ou não. A segunda regra da Figura 4.22 demonstra o caso da expressão `lift` não utilizar esse novo valor, onde todos os sinais s_n utilizados pela expressão obtiveram um resultado `NoChange ()`, portanto o resultado da avaliação também é `NoChange ()`. Já para o caso da expressão `lift` utilizar o novo valor, a terceira regra é aplicada, definindo que existe pelo menos uma variável sinal s_n, sendo utilizada pelo `lift`, com valor `Change` e. A execução da expressão é feita utilizando uma expressão *invoke*, que aplica os novos valores à expressão *lambda*, resultando em um valor `Change` e. O resultado da avaliação de uma expressão `lift`, quando é feita a utilização do valor vindo do evento sendo avaliado, é o valor da nova computação e o ambiente de sinais $Input$, pois diferente do ambiente de variáveis comuns Γ, suas modificações são carregadas para próximas computações.

Para o caso de uma expressão `foldp`, a avaliação é mais simples, pois há somente uma variável sinal que pode sofrer mudanças vindas do GED. A primeira regra da Figura 4.23 demonstra o caso em que o termo sinal da expressão precisa ser reduzido. Já a segunda e terceira regra, assim como o `lift`, demonstram os casos em que a expressão `foldp` utiliza ou não o novo valor. Para o caso do novo valor não ser utilizado, a segunda regra define que o retorno da avaliação é um `Status` de valor `NoChange ()`. Por fim, para o caso do novo valor ser especificamente para a variável sinal utilizada pela expressão ($Input(s) =$`Change` e), a terceira regra define que uma expressão *invoke* deve ser utilizada para aplicar o valor de acc (resultado da avaliação anterior) e o novo valor v à expressão *lambda*. Assim, o resultado da avaliação é um novo valor `Change` e, e o ambiente de sinais, que contém o valor de acc atualizado com o resultado desta computação.

Por fim, a avaliação de uma *thread* cujo termo principal é uma expressão `async` é demonstrada na Figura 4.24. A primeira regra demonstra a redução do termo e, um termo simples, que utiliza variáveis reativa. A segunda e terceira regra, assim como os construtores vistos anteriormente, demonstram a avaliação do termo quando é utilizado ou não o novo valor em $Input$.

Figura 4.24 – Regra de redução para *threads* de termos `async`.

$$\frac{\Gamma, Input \vdash e \to s}{\Gamma, Input \vdash t\langle \textsf{async } e\rangle \longrightarrow t\langle \textsf{async } s\rangle}$$

$$\frac{Input(s) = \textsf{NoChange } ()}{\Gamma, Input \vdash t\langle \textsf{async } s\rangle \longrightarrow \textsf{NoChange } ()}$$

$$\frac{Input(s) = \textsf{Change } e}{\Gamma, Input \vdash t\langle \textsf{async } s\rangle \longrightarrow \textsf{Change } e}$$

Fonte: Autoria própria.

Vale notar que por mais que a avaliação de uma *thread* de um termo `async` seja simples, seu objetivo é executar de forma assíncrona. Isso acontece quando um valor requisitado, pelo termo `async`, se origina de outra *thread*, ou seja, outro termo reativo. Sendo assim, para a simulação de assincronia, basta o termo `async` não esperar pela execução desta outra *thread*, e executar utilizando seu valor anterior. Dessa maneira a assincronia pode existir, levemente modificando a ordem de execução do programa, e seu resultado se mantém próximo ao resultado de uma execução síncrona.

4.6 EXEMPLOS

Alguns exemplos podem ser usados para demonstrar o funcionamento da linguagem, focando principalmente na geração de resultados utilizando entradas reativas e uma possível interação com o paradigma de programação orientada a objeto.

Um exemplo do uso de uma expressão `lift` pode ser visto no Código-fonte 4.1, onde a função anônima `(,)` representa uma função que cria uma tupla a partir de dois valores recebidos. Sempre que qualquer uma das variáveis gerar um novo valor a expressão é executada, resultando em um par `(var,var)`.

Código-fonte 4.1 – Exemplo do uso de uma expressão `lift`.

```
1       lift (,) var var
```

Fonte: Autoria própria.

Como as variáveis de uma expressão `lift` são de tipo `signal` τ, então uma variável que a expressão recebe reativamente pode ser ser outra expressão `lift`. O Código-

fonte 4.2 demonstra uma expressão que retorna uma tupla contendo uma palavra e sua tradução para Francês. Assume-se que a variável `words` gere palavras inteiras vindas de um ambiente externo (entrada do usuário, banco de dados, etc). O `lift` externo tem a mesma funcionalidade do exemplo anterior, criar uma tupla de duas variáveis, já o `lift` interno faz a tradução da mesma palavra recebida para Francês. Nota-se que a execução das duas expressões ocorrem ao mesmo tempo, porém o `lift` externo obrigatoriamente tem que esperar a execução do `lift` interno terminar, para poder gerar seu resultado.

Código-fonte 4.2 – Exemplo de uma expressão `lift` dentro de outra expressão `lift`.

```
1    lift (,) words (lift toFrench words)
```

Fonte: Autoria própria.

Já no Código-fonte 4.3, é apresentado um exemplo de uma expressão `foldp`, onde é feita a contagem de quantas teclas foram apertadas no teclado. Neste exemplo assume-se que uma tecla é processada como um valor inteiro, sendo assim, a função anônima recebe dois valores inteiros e retorna o valor do primeiro (acumulador) após somar com o valor 1.

Código-fonte 4.3 – Exemplo de uma expressão `foldp`.

```
1    foldp (lam x:Int.lam y:Int.(x+1)) 0 Keyboard.PressedKey
```

Fonte: Autoria própria.

A utilização de um termo `async` tem uma maior relevância quando o termo em que é aplicado tem um tempo de execução alto. Porém nem todo exemplo é ideal para a quebra da ordem de execução, pois a corretude do resultado deve ser levada em consideração. O Código-fonte 4.4 apresenta um exemplo do uso de uma expressão `async`, onde o termo a ser executado de forma assíncrona é o mesmo `lift` visto anteriormente.

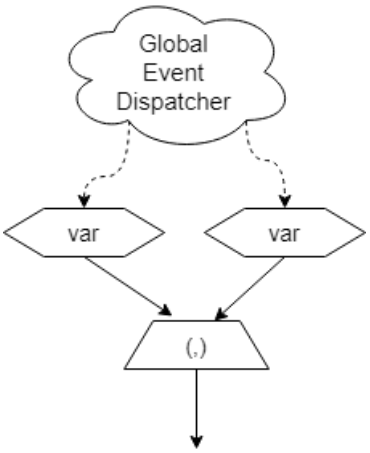
Código-fonte 4.4 – Exemplo de uma expressão `async`.

```
1    let wordPairs = lift (,) words (lift toFrench words)
2    in lift (,) (async wordPairs) Mouse.Position
```

Fonte: Autoria própria.

Neste exemplo a expressão `lift` utiliza o valor da variável `wordPairs` e a posição atual do *mouse* (`Mouse.Position`), e toda vez que qualquer uma dessas variáveis gera um novo valor, a expressão reavalia e gera uma nova tupla. Porém, em um caso em que o processamento de `wordPairs` é significativamente maior que `Mouse.Position`, a expressão `lift` ficaria "presa" esperando o valor de `wordPairs` e não poderia gerar seu resultado

Figura 4.25 – Exemplo de grafo acíclico para uma expressão `lift`.



Fonte: Autoria própria.

rapidamente. A utilização do `async` faz com que essa espera não ocorra, pois toda vez que `Mouse.Position` gera um novo valor, a avaliação de `async wordPairs` resulta no valor de sua avaliação anterior, evitando a espera de um processamento longo.

Uma expressão na linguagem *AsyncRFJ* pode ser visualizada utilizando um grafo acíclico, que contém como elementos o *Global Event Dispatcher* (GED), as variáveis que recebem valores reativamente, e a expressão reativa que gera um resultado. A Figura 4.25 demonstra a visualização de uma expressão `lift` simples vista no Código-fonte 4.1.

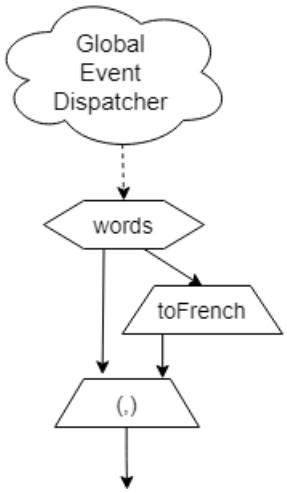
As conexões entre os elementos são importantes, pois demonstra o caminho que o dado percorre durante a avaliação. Por definição, a seta tracejada demonstra valores de tipo sinal, seja um novo valor de variável reativa gerado pelo GED, ou o valor resultante de uma operação salva em uma variável, podendo ser usada pelo operador `async`.

Utilizando o exemplo do Código-fonte 4.2, onde é definida uma expressão `lift` que usa o valor resultante de outra expressão `lift`, é possível criar grafo apresentado na Figura 4.26. Neste grafo é possível ver que o valor de `words`, que vem do GED, é propagado nas duas operações `lift` ao mesmo tempo. Porém, como a avaliação de `(,)` depende do valor de `toFrench`, a avliaação de `(,)` bloqueia e espera o processamento completo de `toFrench`.

Por fim, para visualizar a diferença de execução de um termo `async`, usa-se o exemplo do Código-fonte 4.4 para gerar os grafos da Figura 4.27. O grafo `(a)` representa a execução do programa utilizando o `async` na variável `wordPairs`, já o grafo `(b)` representa o mesmo termo sendo executado de uma maneira completamente síncrona.

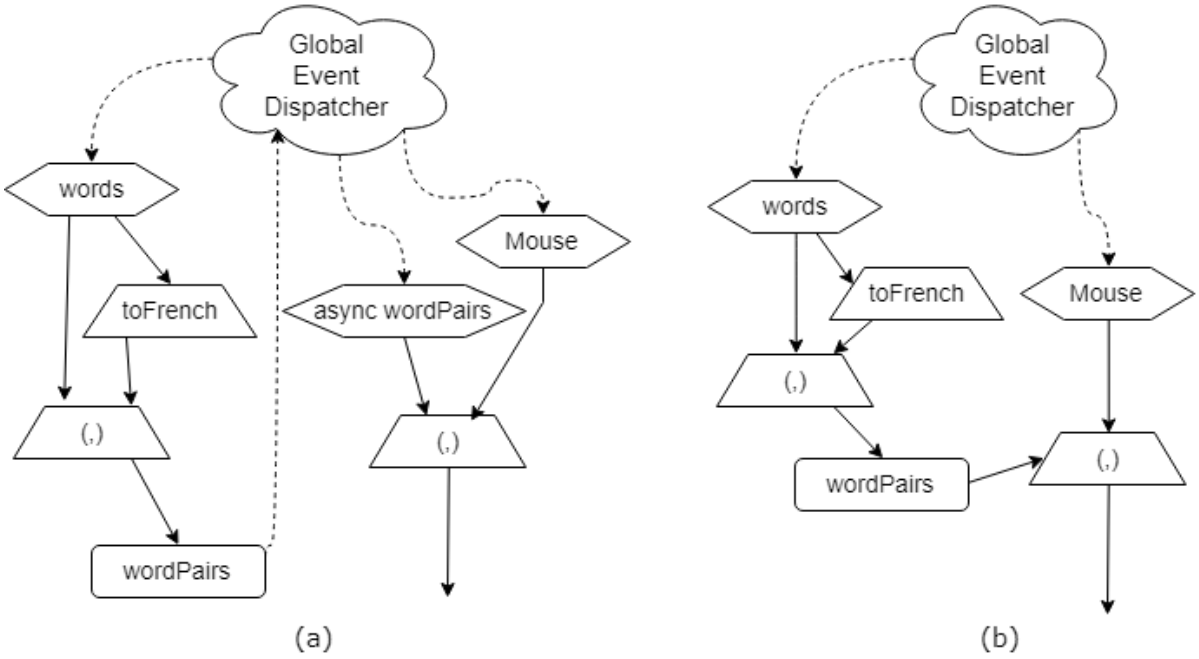
A principal diferença nas execuções da Figura 4.27 está no valor que será atribuído à variável `wordPairs`. No caso do grafo `(a)`, por utilizar o operador `async`, a variável `wordPairs` é tratada como uma variável vinda diretamente do GED. Isso faz com que o valor atual da variável esteja guardado no ambiente de sinais $Input$, se tornando um valor

Figura 4.26 – Exemplo de grafo acíclico para uma expressão com dois operadoes `lift`.



Fonte: Autoria própria.

Figura 4.27 – Exemplo de grafo acíclico para uma expressão com e sem o uso do `async`.



Fonte: Autoria própria.

de fácil acesso, e não precisando re-executar toda a expressão.

No caso da Figura 4.27 `(a)`, quando um novo valor para a variável sinal `words` é gerado, a expressão `lift (,) (async wordPairs) Mouse` não re-executa, pois `words` não extá presente nos sinais utilizados (`wordPairs` e `Mouse`). Porém a expressão `wordPairs = lift (,) words (lift toFrench words)`, recebe esse novo valor e gera uma tupla que irá ser guardada na variável `wordPairs`. Dessa maneira, um novo evento occore, pois um novo valor para `wordPairs` foi gerado, acionando o `lift` que contém o termo sinal `async`

`wordPairs`, que inicia uma nova computação para a expressão, e gera uma nova tupla `(wordPairs, Mouse)`.

Já no caso de um novo valor para a variável `Mouse`, o grafo `(a)` não precisa recalcular o valor da variável `wordPairs`, pois é utilizada em um termo `async`, economizando tempo de execução do programa. Assim, utilizando o valor anterior de `wordPairs`, a expressão do grafo `(a)` gera a nova tupla `(wordPairs,Mouse)` como resultado.

Já no caso do grafo `(b)`, quando qualquer uma das duas variáveis utilizadas ,`wordPairs` ou `Mouse`, gera um novo valor, é feita a reavaliação da expressão completa. Pode-se dizer que o resultado é sempre o mais correto possível, porém o tempo para a execução é maior, e isso pode se tornar prejudicial a um programa que realiza um alto número de eventos (por exemplo, `Mouse` receber novos valores constantemente).

5 UM INTERPRETADOR PARA ASYNCRFJ

Utilizando as definições apresentadas nos capítulos anteriores, foi desenvolvido um interpretador para a linguagem *AsyncRFJ*[1], que leva em consideração as características da proposta Featherweight Java, e as extensões mencionadas anteriormente. O objetivo principal é viabilizar a utilização do conceito funcional reativo de modo a possibilitar a utilização de construtores reativos em uma linguagem orientada a objetos.

Para a implementação do interpretador deste trabalho, foi utilizada a linguagem de programação *Haskell*, que é uma avançada linguagem de programação puramente funcional, baseada no cálculo lambda, polimorficamente e estaticamente tipada, com avaliação *lazy* e funcionalidade de *pattern matching*. Muitas das funcionalidades fornecidas pela linguagem auxiliam no processo de manipulação e interpretação das árvores de sintaxe abstratas (AST). Para este interpretador, foi utilizado como base o sitema de tipos e semântica de Feitosa (2018), que define um interpretador básico especificamente para o Featherweight Java, porém, o analisador léxico foi baseado no trabalho de Feitosa (2016), que faz a utilização do *Happy*, um gerador de *parser* para a linguagem *Haskell*.

5.1 ANALISADOR LÉXICO

A função do analisador léxico é converter o código-fonte da linguagem para uma lista de *tokens*. A Tabela 5.1 apresenta as palavras chaves reservadas na linguagem, e seu *token* correspondente.

Tabela 5.1 – Palavras reservadas da linguagem.

Palavra-Chave	**Token**	**Palavra-Chave**	**Token**
class	TokenKWClass	let	TokenKWLet
extends	TokenKWExtends	in	TokenKWIn
super	TokenKWSuper	boolean	TokenKWBoolean
this	TokenKWThis	true	TokenKWTrue
new	TokenKWNew	false	TokenKWFalse
return	TokenKWReturn	int	TokenKWInt
Object	TokenKWObject	string	TokenKWString
if	TokenKWIf	lift	TokenKWLift
else	TokenKWElse	foldp	TokenKWFoldp
invoke	TokenKWInvoke	async	TokenKWAsync

Fonte: Autoria própria.

[1] Implementação completa pode ser encontrada em: <https://github.com/diogocrds/AsyncRFJ>

A linguagem também aceita uma série de caracteres especiais, apresentados na Tabela 5.2. Além disso, pode-se usar números inteiros para operações matemáticas binárias, e nomes para representar classes, atributos, variáveis, métodos, etc.

Tabela 5.2 – Caracteres especiais da linguagem.

Palavra-Chave	Token
‘{’	TokenLBrace
‘}’	TokenrBrace
‘(’	TokenLParen
‘)’	TokenRParen
‘,’	TokenComma
‘;’	TokenSemi
‘.’	TokenDot
‘=’	TokenAssign

Palavra-Chave	Token
‘<’	TokenLT
‘>’	TokenGT
‘->’	TokenArrow
‘*’	TokenTimes
‘/’	TokenDivide
‘+’	TokenPlus
‘-’	TokenMinus
‘==’	TokenEqual
‘”’	TokenQuote

Fonte: Autoria própria.

Tabela 5.3 – Nomes ou números definidos pelo usuário.

Palavra-Chave	Token
name	TokenName $$
number	TokenrNumber $$

Fonte: Autoria própria.

O analisador léxico faz a leitura de cada caracter escrito no código-fonte do programa, utilizando a função *lexer*, que é responsável por gerar uma lista de *tokens* na saída do processo. O Código-fonte 5.1 apresenta um trecho da função *lexer*.

Código-fonte 5.1 – Trecho de código-fonte do analisador léxico.

```
1  lexer :: String -> [Token]
2  lexer [] = []
3  lexer ('{':cs) = TokenLBrace : lexer cs
4  lexer ('}':cs) = TokenRBrace : lexer cs
5  lexer ('(':cs) = TokenLParen : lexer cs
6  lexer (')':cs) = TokenRParen : lexer cs
7  lexer (',':cs) = TokenComma : lexer cs
8  lexer (';':cs) = TokenSemi : lexer cs
9  lexer ('.':cs) = TokenDot : lexer cs
10 lexer ('+':cs) = TokenPlus : lexer cs
11 lexer ('/':cs) = TokenDivide : lexer cs
```

```
12 lexer ('*':cs) = TokenTimes : lexer cs
13 lexer ('"':cs) = TokenQuote : lexer cs
14 lexer (c:cs)    | isSpace c = lexer cs
15                 | isAlpha c = lexStr (c:cs)
16                 | isDigit c = lexDigit (c:cs)
17                 | isToken c = lexSymbol (c:cs)
```

Fonte: Autoria própria.

Como demonstrado no código acima, a função faz a coversão direta de caracteres especiais da linguagem, e utiliza funções auxiliares para tratar de pontos específicos, como leitura de *strings*, dígitos e símbolos. Este processo faz com que o interpretador somente aceite caracteres permitidos pela linguagem.

5.2 ANALISADOR SINTÁTICO

A função do analisador sintático é realizar o *parsing* do programa, neste processo é feita uma analise na sequência de *tokens* de entrada para verificar sua estrutura gramatical de acordo com uma gramática formal, que é expressa através da notação *BNF*.

Para este trabalho foi utilizado um gerador de analisadores sintáticos, chamado *Happy*, da própria linguagem *Haskell*. É semelhante à ferramentas como *yacc* para a linguagem C e *AntLR*, e foi utilizado para facilitar a construção do *parser* para a linguagem. A ferramenta funciona utilizando um arquivo onde é definida a especificação da gramática *BNF*, e produz como saída um módulo *Haskell* contendo o *parser* para a gramática desejada.

O *parsing* do programa acontece quando o analisador sintático manipula a lista de *tokens* gerados pelo analisador léxico, transformando a lista em uma *AST*. Nesta etapa é verificado se não há erros na escrita do programa, ou seja, se está sintaticamente correto, respeitando as regras gramaticais definidas. Para um programa FJ com suporte a programação reativa, a definição inclui uma lista de classes, uma lista de sinais de entrada ($Input$), a lista de eventos ocorridos, e o termo principal a ser avaliado. O Código-fonte 5.2 apresenta a defingão da gramática de um programa escrito na linguagem *AsyncRFJ*.

Código-fonte 5.2 – Trecho da gramática *BNF* que representa o programa sendo interpretado.

```
1 Program : ClassList InputList EventList Term ';'    { Program $1 $2 $3 $4 }
2
3 data Program = Program [(String,ClassDef)] [(String, Term, Status)]
4                        [(String, Term)] Term
5              deriving (Show, Eq)
```

Fonte: Autoria própria.

A lista de classes contém uma ou mais classes em seu conteúdo, e após o processo de análise sintática é transformada na estrutura chamada *tabela de classes*, sendo utilizada em todo o processo de análise semântica. A tabela de classes contém tuplas de uma *string*, que representa o nome da classe, e uma estrutura *ClassDef*, que contém todas as definições internas de uma classe (atributos, contrutores e métodos). A descrição da gramática de classes fornecida para o *Happy* pode ser vista no Código-fonte 5.3.

Código-fonte 5.3 – Trecho da gramática *BNF* para as classes da linguagem.

```
1 ClassList       : ClassDef                { [$1] }
2                 | ClassList ClassDef      { $2 : $1 }
3
4 ClassDef : class ClassName extends ClassName '{'
5                 AttrList ConstrDef MethodList '}'
6             { ($2,ClassDef $2 $4 $6 $7 $8) }
```

Fonte: Autoria própria.

O módulo em *Haskell*, gerado pelo *Happy*, processa a lista de *tokens* vinda do analisador léxico e faz uma transformação utilizando construtores de tipo previamente definidos, e o resultado é a árvore de sintaxe abstrata para o código sendo processado. O Código-fonte 5.4 apresenta os construtores de tipo para classes, construtores e métodos.

Código-fonte 5.4 – Construtores da árvore de sintaxe abstrata da linguagem.

```
1 data ClassDef = ClassDef String String [(Type,String)] ConstrDef [MethodDef]
2                 deriving (Show, Eq)
3
4 data ConstrDef = ConstrDef String [(Type,String)] [String] [(String,String)]
5                 deriving (Show, Eq)
6
7 data MethodDef = MethodDef Type String [(Type,String)] Term
8                 deriving (Show, Eq)
```

Fonte: Autoria própria.

O analisador sintático verifica a criação da tabela de classes, utilizando as definições acima, e também verifica a construção do termo a ser avaliado. Este termo representa o ponto de partida da execução do programa fornecido, e pode ser descrito utilizando diferentes construtores como visto no Código-fonte 5.5, que apresenta alguns dos construtores da permitidos pela linguagem.

Código-fonte 5.5 – Trecho da gramática *BNF* para os termos da linguagem.

```
1 Term : BooleanLiteral                     { BooleanLiteral $1 }
2      | name                               { Var $1 }
3      | number                             { Int $1 }
4      | '"' name '"'                       { Str $2 }
```

```
5       | this '.' name                                  { ThisAccessAttr $3 }
6       | this '.' name '(' TermList ')'                { ThisAccessMeth $3 $5 }
7       | Term '.' name                                 { AttrAccess $1 $3 }
8       | Term '.' name '(' TermList ')'                { MethodAccess $1 $3 $5 }
9   -- Construtores Reativos
10      | lift '('('(' ParamList ')' "->" Term ')' '(' TermList ')'
11                                                      { Lift $4 $7 $10 }
12      | foldp '('('(' ParamList ')' "->" Term ')' Term '(' Term ')'
13                                                      { Foldp $4 $7 $9 $11 }
14      | async Term                                    { Async $2 }
```

Fonte: Autoria própria.

Todas as construções da gramática *BNF* apresentadas acompanham um construtor de tipo associado (*AST*), e sua descrição completa está disponível no Apêndice B.

5.3 ANALISADOR SEMÂNTICO

A próxima etapa do interpretador é a análise semântica do programa de entrada, e esta etapa tem a função de dar significado e realizar as verificações de tipo das construções de alto nível da linguagem. Neste ponto, o código está sintaticamente correto, ou seja, respeita as regras de construção do código-fonte, e o a(s) AST(s) foram criadas para o código correspondente.

Baseando-se na definição do FJ, um programa que utiliza o paradigma OO contém uma conjunto de classes e um termo como ponto de entrada. Para a linguagem proposta, também é feita a definição de uma tabela de classes e o termo, porém é nesta fase da interpretação do programa que os dois novos componentes são utilizados: a lista de sinais de entrada e a lista de eventos reativos que serão processados.

5.3.1 Funções Auxiliares

Como visto nas regras definidas pelo FJ original, é necessário definir diversas funções auxiliares que são usadas nas regras de avaliação, e manipulação da *tabela de classes*. Dentre estas definições algumas funcionalidades são: verificação de *subtipos*, obtenção de atributos, nome, tipo de retorno, e obtenção do corpo de métodos de uma determinada classe.

A verificação de subtipos reflete a definição da Figura 3.2, definindo se uma classe é subtipo de outra classe, utilizando o conteúdo da *tabela de classes*. Esta função tem como objetivo auxiliar o processo de verificação de tipo e interpretação de termos que envolvem o mecanismo de herança. O Código-fonte 5.6 apresenta a definição da função `subtyping`.

Código-fonte 5.6 – Código para a função *subtyping*.

```
1 subtyping :: String -> String -> CT -> Bool
2 subtyping _ "Object" _ = True
3 subtyping "Object" _ _ = False
4 subtyping c c' ct
5     | c == c' = True
6     | otherwise = case (Data.Map.lookup c ct) of
7                     Just (ClassDef _ c'' _ _ _) ->  if c' == c'' then True
8                                                     else subtyping c'' c' ct
9                     _ -> False
```

Fonte: Autoria própria.

Outra função auxiliar presente no interpretador, é a função chamada `fields`, que acessa uma determinada classe e retorna seus atributos e seus respectivos tipos, seguindo as regras definidas na Figura 3.3. Além disso, a função também percorre as *superclasses* definidas, retornando também todos os atributos herdados pela classe alvo.

Código-fonte 5.7 – Código para a função *fields*.

```
1 fields :: String -> CT -> Maybe [(Type, String)]
2 fields "Object" _ = Just []
3 fields c ct = case (Data.Map.lookup c ct) of
4                 Just (ClassDef _ c'' attrs _ _) ->
5                   case (fields c'' ct) of
6                     Just base -> Just (base ++ attrs)
7                     _ -> Nothing
8                 _ -> Nothing
```

Fonte: Autoria própria.

As funções que tratam da obtenção dos métodos de uma classe auxiliam no processo de interpretação da invocação de métodos, e também da avaliação dos tipos. Para essas situações, foram modeladas no interpretador as funções *methods* e *mtype*, implementando as regras apresentadas anteriormente na Figura 3.3 e na Figura 3.5 respectivamente.

5.3.2 Verificação de Tipos

Esta etapa faz a verificação da coerência de tipos utilizados no programa. É preciso fazer a verificação de atributos, métodos, construção de classes e termos, para garantir que o programa execute sem erros de tipo (*type-safe*), ou seja, garantir que ocorra somente a utilização de valores de tipos suportados. Na implementação deste trabalho, a verificação de tipos ocorre antes da avaliação do programa, assim durante a avaliação do termo existe a garantia de que as classes estão bem formadas e o programa só utiliza tipos válidos.

A verificação de tipos da *tabela de classes* se dá pelo uso da função `checkCTable`, que faz a checagem de cada classe na lista utilizando as regras vistas anteriormente.

Código-fonte 5.8 – Código para a função *checkCTable*.

```
1  checkCTable :: [(String, ClassDef)] -> [(String, Term, Status)] -> Bool
2  checkCTable ct i =
3              (Data.List.all
4                  (\(c,cl) ->
5                    classTyping cl Data.Map.empty (Data.Map.fromList ct)
                         (Data.Map.fromList (mapInputList i))) ct)
6
7  mapInputList :: [(String, Term, Status)] -> [(String, (Term, Status,Int))]
8  mapInputList i = Data.List.map (\(x,y,z) -> ( x,(y,z,-1) )) i
```

Fonte: Autoria própria.

A tipagem de cada classe verifica se a mesma está bem definida, o que inclui a verificação de seus atributos, construtor e métodos declarados. Os atributos podem ser de algum tipo aceito pela linguagem ou de um tipo definido no código-fonte, sendo processado como uma classe. O construtor inicia todos os atributos da classe, e os valores recebidos são de tipo correspondente para cada atributo. Já para os métodos declarados na classe, é preciso garantir que somente tipos válidos sejam usados, seja em seus atributos, no corpo do método ou em seu retorno definido. O Código-fonte 5.9 apresenta a checagem de tipo de uma classe.

Código-fonte 5.9 – Código para a função *classTyping*.

```
1  classTyping :: ClassDef -> Env -> CT -> Input -> Bool
2  classTyping cl@(ClassDef c b attrs (ConstrDef cn pc s ths) meths) ctx ct i =
3    case (fields b ct) of
4      Just flds ->
5        if (pc == (flds ++ attrs)) then
6          if (Data.List.all (\(n',n'') -> n' == n'') ths) then
7            let p' = Data.List.map (\(tp, nm) -> nm) pc
8                p'' = s ++ (Data.List.map (\(n',n'') -> n') ths)
9              in (p' == p'') && (Data.List.all (methodTyping ctx ct cl i) meths)
10         else
11           False
12       else
13          False
14     _ -> False
```

Fonte: Autoria própria.

É possível observar que a função `classTyping` faz a checagem se os atributos são corretos, levando em consideração atributos herdados obtidos utilizando a função auxiliar `fields`. Quando não houver nenhum erro na declaração dos atributos, é utilizada outra função para a verificação de métodos, chamada `methodTyping`.

Como visto anteriormente, um programa na linguagem *AsyncRFJ* contém uma lista de declarações de classes (*Class Table*), uma lista de sinais de entrada (*Input*), uma lista de eventos reativos (*GED*) e um termo principal a ser avaliado. Após checagem de tipos da *tabela de classe*, é feito diretamente a checagem de tipos do termo principal. Pois a lista de sinais e a lista de eventos são somente tuplas onde um nome é atribuído a um valor, e se já estão sintaticamente corretas, não há erro de tipo.

Assim como no FJ, o termo principal é o início da execução do programa. É neste termo que ocorrerá a criação de objetos, acesso à atributos, métodos, etc. Portanto é necessário verificar o tipo de cada um dos termos possíveis. A função `checkTerm` aplica a função recursiva `typeOf` ao termo principal, e a verificação de seu retorno é avaliado como tipo válido ou não. Vale notar que é possível capturar erros tipagem neste ponto da execução, como variável não encontrada ou número incorreto de parâmetros em um método, etc.

Código-fonte 5.10 – Código para a função *checkTerm*.

```
1  checkTerm :: [(String, ClassDef)] -> [(String, Term, Status)] -> Term -> Bool
2  checkTerm ct i t =
3      case (typeof Data.Map.empty (Data.Map.fromList ct) (Data.Map.fromList
           (mapInputList i)) t) of
4          Right (TypeClass t) -> True
5          Right (TypeBool) -> True
6          Right (TypeInt) -> True
7          Right (TypeString) -> True
8          Right (TypeClosure e t) -> True
9          Right (SignalType (TypeClass t)) -> True
10         Right (SignalType TypeBool) -> True
11         Right (SignalType TypeString) -> True
12         Left (VariableNotFound e) -> error "Error: Declaration not Found."
13         Left (ParamsTypeMismatch e) -> error "Error: Param miss-match."
14         Left (ClassNotFound e) -> error "Error: Class not Found."
15         Left (FieldNotFound e) -> error "Error: Field Not Found."
16         Left (MethodNotFound e s) -> error "Error: Method Not Found."
17         t -> error ("Error: "++(show t))
```

Fonte: Autoria própria.

Em conjunto aos tipos primários possíveis de um programa (`TypeClass`, `TypeInt`, `TypeBool`, `TypeString`, etc), há também sua versão reativa (`SignalType`), um termo é aceito se for de um desses tipos. Nota-se que para acomodar a captura de erros de tipo, a função `typeof` retorna uma estrutura de dado `Either`, onde o lado esquerdo é atribuído a um erro de tipo, e o lado direito é atribuído ao um tipo aceito pela linguagem.

Como uma maneira de fazer a verificação de tipo de um termo, foi definida a função `typeof` (Código-fonte 5.11), que retorna o tipo de uma expressão utilizando os contextos Γ e $Input$. Na implementação, esses dois contextos são representados por uma lista de tu-

plas, contendo os termos acessíveis naquele momento e seus respectivos tipos. No início do programa, o contexto Γ é vazio, e é preenchido pelas variáveis repassadas como parâmetros para funções ou métodos, e variáveis de tipo simples declaradas em expressões `let`. Já o contexto $Input$, no início do programa, contém as variáveis que serão tratadas como entrada de dados reativos (*input signals*), e são modificadas somente durante a etapa de avaliação.

Código-fonte 5.11 – Código para a função *typeof*.

```
1  typeof :: Env -> CT -> Input -> Term -> Either TypeError Type
2  typeof ctx ct input (Int i) = Right (TypeInt)
3  typeof ctx ct input (Str i) = Right (TypeString)
4  typeof ctx ct input (BooleanLiteral t) = Right (TypeBool)
5  typeof ctx ct input (Var v) =                          -- T-Var
6    case (Data.Map.lookup v input) of
7      Just (Var t,_,_) -> Right (SignalType (TypeClass v))
8      Just (BooleanLiteral t,_,_) -> Right (SignalType TypeBool)
9      Just (Int t,_,_) -> Right (SignalType TypeInt)
10     Just (Str t,_,_) -> Right (SignalType TypeString)
11     _ -> case (Data.Map.lookup v ctx) of
12            Just var -> return var
13            _ -> throwError (VariableNotFound v)
14 typeof ctx ct input (Let v t1 t2) =                    -- T-Let
15   let t1' = typeof ctx ct input t1
16       ctx' = Data.Map.insert v (getRight t1') ctx
17   in typeof ctx' ct input t2
18 typeof ctx ct input (ClosureDef p t) =                 -- T-Lam
19   let p' = Data.List.map (\(t,n) -> (n,t)) p
20       tp' =  (Data.List.map (\(t,n) -> t) p)
21       ctx' = Data.Map.union (Data.Map.fromList p') ctx in
22     case (typeof ctx' ct input t) of
23       Right t -> Right (TypeClosure t tp')
24 typeof ctx ct input (InvokeClosure c@(ClosureDef p t1) t) = -- T-Invok
25   if ((Data.List.length p)==(Data.List.length t)) then
26     let p'=Data.List.zipWith (\(tp,_) tE -> (tp,(typeof ctx ct input tE))) p t
27     in if (Data.List.all (\(p1,p2) -> (p1==(getRight p2))) p') then
28          case (typeof ctx ct input c) of
29            Right (TypeClosure r p) ->  Right r
30        else error ("Closure: miss-match type of parameters"++(show p'))
31   else error "Closure: params miss-match"
```

Fonte: Autoria própria.

Pode-se observar que a função `typeof` faz uma avaliação recursiva do termo para a obtenção de seu tipo. No caso da avaliação de um termo valor (*objeto*, *booleanos*, *inteiros* e *strings*), seu tipo é retornado diretamente. Para o caso de uma variável (`Var v`), primeiamente é feita a consulta no ambiente de sinais $Input$, se não houver um re-

torno válido, é feita a consulta no ambiente de variáveis simples Γ. A checagem de tipo de uma expressão `let` faz uma modificação no ambiente Γ, adicionando a nova variável sendo declarada, e faz a checagem de seu termo interno utilizando este novo ambiente Γ. Da mesma maneira, a checagem de tipo de uma expressão que define uma *closure* (`ClosureDef`), adiciona suas variáveis parâmetro no ambiente Γ e checa seu termo inteno, retornando um contrutor que contém uma lista de tipo dos parâmetros, e o tipo de seu termo interno. Já a expressão que checa a execução de uma *closure* (`InvokeClosure`), verifica se o número de parâmetros recebidos está correto e se o tipo de cada um está corretamente associado, retornando o tipo do retorno da função anônima.

Para o caso dos contrutores reativos (`lift`, `foldp` e `async`), as regras de tipo são demonstradas no Código-fonte 5.12. No caso de um `lift`, é feita a verificação se cada termo recebido tem o tipo correspondente aos parâmetros da expressão *lambda*. No caso de um `foldp`, a variável *acumulador* é tratada como uma variável no contexto Γ, a relação de tipos entre termos recebidos e parâmetros também é verificada. Já no caso de uma expressão `async`, é simplesmente retornado o tipo de seu termo interno.

Código-fonte 5.12 – Trecho do código para a função *typeof* para construtores reativos.

```
1  typeof :: Env -> CT -> Input -> Term -> Either TypeError Type
2  typeof ctx ct input (Lift p e t) = -- T-Lift
3      if ((Data.List.length p)==(Data.List.length t)) then
4        let p' = Data.List.zipWith (\(tp,_) tE -> (tp,(typeof ctx ct input tE)))
             p t in
5          if (Data.List.all (\(p1,p2) -> (p1==(getRight p2))) p') then
6            let ctx' = Data.Map.union (Data.Map.fromList (Data.List.map (\(t,n)
                 -> (n,t)) p)) ctx
7            in case (typeof ctx' ct input e) of
8              Right t -> Right (SignalType t)
9          else error "Lift: miss-match type of parameters"
10     else error "Lift: miss-matched number of parameters"
11 typeof ctx ct input (Foldp p e acc t) = -- T-Foldp
12     if ((Data.List.length p)==2) then
13     let p' = Data.List.zipWith (\(tp,_) tE -> (tp,(typeof ctx ct input tE))) p
           ([t]++[acc]) in
14       if (Data.List.all (\(p1,p2) -> (p1==(getRight p2))) p') then
15         let ctx' = Data.Map.union (Data.Map.fromList (Data.List.map (\(t,n) ->
               (n,t)) p)) ctx
16         in case (typeof ctx' ct input e) of
17           Right t -> Right (SignalType t)
18       else error "Foldp: miss-match type of parameters"
19     else error "Foldp: miss-matched number of parameters"
20 typeof ctx ct input (Async e) = typeof ctx ct input e -- T-Async
```

Fonte: Autoria própria.

Com a validação da tabela de classes e o termo principal, esta etapa finaliza e as

ASTs são repassadas para a etapa de avaliação dos termos. Vale notar que em caso de algum erro na verificação de tipos durante qualquer uma das checagens vistas anteriormente, o programa é abortado imediatamente.

5.3.3 Avaliação dos Termos

O processamento do interpretador finaliza com a avaliação do termo do programa. A linguagem proposta é baseada no FJ, que é uma visão funcional do Java sem efeitos colaterais, portanto o funcionamento da avaliação dos termos é similar ao processamento de linguagens funcionais. Com a utilização do *Haskell*, a implementação das funções que realizam a redução do termo é muito próxima às regras semânticas definidas nos capítulos anteriores.

As regras definidas na semântica da linguagem podem ser interpretadas como funções parciais que, quando aplicadas a um termo que não é um valor, resultam em um novo termo representando o próximo passo de avaliação. Já no caso da função ser aplicada a um valor, a avaliação para, significando que não há um próximo passo a partir do termo atual e possivelmente o processo do interpretador está finalizado.

Para realizar a interpretação do programa escrito, foi definida a função `eval`, que avalia o termo do programa até que se torne um valor final. O Código-fonte 5.13 apresenta a definição da função `eval`, vale notar que a avaliação é feita por uma função secundária `eval'`, e também, é feito o uso de funções auxiliares como `isValue` e `isSignalValue`. Os parâmetros necessários para a avaliação de um termo são: o ambiente global de variáveis simples (`ctx`), a tabela de classes (`ct`), a lista de variáveis reativas $Input$, o número de *threads* geradas até o momento (`numT`), e o termo em sendo avaliado.

Código-fonte 5.13 – Código para a função *eval*.

```
1  eval :: EnvG -> CT -> Input -> Int -> Term -> EProgram
2  eval ctx ct i numT e =
3    case (eval' ctx ct i numT e) of
4      (EProgram (Just e') t) ->
5        if ((isValue ct e') || (isSignalValue e')) then EProgram (Just e') t
6        else
7          case eval ctx ct i numT e' of
8            EProgram e'' t' -> EProgram e'' (t++t')
9      (EProgram Nothing t) -> EProgram (Just e) t
```

Fonte: Autoria própria.

Durante toda a avaliação do programa, o termo tratado como resultado de uma avaliação é uma estrutura de dados `EProgram`, que representa um programa avaliado (Evaluated Program). Esta estrutura contém um termo, resultante da avaliação feita, e uma lista

de *threads* geradas durante a avaliação, podendo ser vazia.

5.3.3.1 Avaliação Funcional

O processo de avaliação realizado pelo interpretador é dividido em duas partes, a avaliação funcional e a avaliação de sinais. A primeira, avaliação funcional, faz o uso da função `eval'`, que se difere da função `eval` vista acima por não testar se o valor é final. Um trecho da função `eval'` pode ser visto no Código-fonte 5.14.

Código-fonte 5.14 – Trecho do código para a função *eval'*.

```
1  eval' :: EnvG -> CT -> Input -> Int -> Term -> EProgram
2  eval' ctx ct input numT (Var v) =
3    case (Data.Map.lookup v input) of
4      Just (v',_,_) -> EProgram (Just (SignalTerm v')) []
5      Nothing -> EProgram (Data.Map.lookup v ctx) []
6  eval' ctx ct input numT (Let v e1 e2) =
7    let e1' = eval' ctx ct input numT e1 -- R-Let
8      in case e1 of
9          (Lift p e t) ->
10           case e1' of
11           (EProgram (Just (SignalTerm e'')) t') ->
12             let input' = (Data.Map.insert v (e'',NoChange e'',numT+1) input)
13             in case (eval' ctx ct input' (numT+(length t')) e2) of
14                 (EProgram (Just e2') t'') -> EProgram (Just e2') (t'++t'')
15         (Foldp p e acc t) -> [..]
16         (Async e) -> [..]
17         _ -> case e1' of
18              (EProgram (Just (SignalTerm e)) t) ->
19                let input' = (Data.Map.insert v (e,NoChange e,-1) input)
20                in case (eval' ctx ct input' numT e2) of
21                  (EProgram e2' t') -> EProgram e2' (t++t')
22              (EProgram (Just e) t) -> let ctx' = (Data.Map.insert v e ctx)
23                                       in (eval ctx' ct input numT e2)
24 eval' ctx ct input numT (InvokeClosure (ClosureDef par e) t) ==
25     let p' = Data.List.zipWith (\(ty,var) (val) ->
26           case (eval' ctx ct input numT val) of
27             (EProgram (Just val') t) -> (var,val')) par t
28         ctx' = Data.Map.union (Data.Map.fromList p') ctx
29     in eval' ctx' ct input numT e
```

Fonte: Autoria própria.

No código acima é possível ver alguns dos pontos importantes da avaliação de um programa AsyncRFJ. Para o caso de uma variável (`Var v`), primeiro, é feita a checagem no ambiente de sinais $Input$ e caso não seja encontrado um valor correspondente, é feita

a checagem no ambiente de variáveis simples Γ. É importante fazer a checagem no ambiente $Input$ primeiro, para que variáveis de sinais possam ser diferenciadas de variáveis normais, se tornando uma espécie de palavra-chave da linguagem.

Para o caso de uma expressão `let`, existem duas situações que devem ser avaliadas de maneiras diferentes: o valor de e_1 (em `let x = e1 in e2`) ser de tipo sinal ou não. Na situação em que e_1 não é de tipo sinal, a variável é simplesmente adicionada ao ambiente Γ e o próximo passo de avaliação é avaliar o termo e_2, utilizando o ambiente modificado Γ'. Já para o caso de e_1 ser de tipo sinal, porém não é uma expressão reativa, a variável é adicionada ao ambiente $Input$, e da mesma maneira, a avaliação continua com o termo e_2. Por fim, quando e_1 for uma expressão reativa `lift`, `foldp`, ou `async`, é preciso levar em consideração o número de *threads* geradas, portanto, a variável é adicionada no ambiente $Input$, e a avaliação de e_2 recebe o número atual de *threads* somado de 1. Vale notar que o único momento em que novos valores são adicionados ao ambiente $Input$ é na avaliação de expressões `let`, `foldp` e `async`.

Os campos de uma variável no ambiente $Input$ são: um identificador, que corresponde ao nome da variável que será chamada; um valor salvo que representa seu valor inicial; um valor utilizando a estrutura $Status$ (podendo ser $Change\ e$ ou $NoChange\ e$), que será usado para avaliações reativas; e um número referente à *thread* em que o valor se origina.

A última regra do Código-fonte 5.14 é a regra referente a uma execução de uma *closure*, definida como `InvokeClosure (ClosureDef par e) t`. A avaliação simplesmente adiciona todos os valores do termo `t` ao abiente Γ, associando-os aos parâmetros `par` da *closure* a ser executada. Com este novo ambiente Γ', o próximo passo da avaliação é avaliar o termo e.

Como definido anteriormente, o único momento durante a avaliação do programa em que novas *threads* são criadas, é durante a avaliação de expressões reativas. Os Códigos-fonte 5.15, 5.16 e 5.17 apresentam, respectivamente, o trecho da função `eval'` que avalia expressões `lift`, `foldp` e `async`, criando suas respectivas *threads* e retornando o valor resultado da avaliação.

Código-fonte 5.15 – Trecho do código da função *eval'* o construtor `lift`.

```
1  eval' :: EnvG -> CT -> Input -> Int -> Term -> EProgram
2  eval' ctx ct input numT l@(Lift p e t) =                    -- R-Lift
3    let t' = (Data.List.map (\x -> case (eval' ctx ct input (numT+1) x) of
4               EProgram (Just x') r -> r) t)
5        res = (eval' ctx ct input (numT+1+(length t'))
6               (InvokeClosure (ClosureDef p e) t))
7    in case res of
8        (EProgram (Just (SignalTerm e')) t'') ->
9          let tList = (Thread (numT+1) ctx ct input l) in
10           EProgram (Just (SignalTerm e')) ([tList]++(Data.List.concat t')++t'')
```

```
11        (EProgram (Just e') t'') ->
12          let tList = (Thread (numT+1) ctx ct input l) in
13            EProgram (Just (SignalTerm e')) ([tList]++(Data.List.concat t')++t'')
```

Fonte: Autoria própria.

Código-fonte 5.16 – Trecho do código da função *eval'* o construtor `foldp`.

```
1  eval' ctx ct input numT f@(Foldp p e acc t) =              -- R-Foldp
2    let t' = case (eval' ctx ct input (numT+1) t) of (EProgram ev tr) -> tr
3        res' = (eval' ctx ct input (numT+1+(length t'))
4                (InvokeClosure (ClosureDef p e) ([t]++[acc])))
5        input' = case res' of
6                   (EProgram (Just rres) tres) ->
7                     (Data.Map.insert "acc" (rres,NoChange rres,-1) input)
8    in case res' of
9        (EProgram (Just (SignalTerm e')) t'') ->
10         let tList = (Thread (numT+1) ctx ct input' f)
11         in EProgram (Just (SignalTerm e')) ([tList]++t'++t'')
12       (EProgram (Just e') t'') ->
13         let tList = (Thread (numT+1) ctx ct input' f)
14         in EProgram (Just (SignalTerm e')) ([tList]++t'++t'')
```

Fonte: Autoria própria.

Código-fonte 5.17 – Trecho do código da função *eval'* o construtor `async`.

```
1  eval' ctx ct input numT a@(Async e) =                      -- R-Async
2    let t' = case (eval' ctx ct input (numT+1) e) of (EProgram ev tr) -> tr
3        res' = (eval' ctx ct input (numT+1+(length t')) e)
4        input' = case res' of (EProgram (Just rres) tres) ->
5          (Data.Map.insert "async" (rres,NoChange rres,-1) input)
6    in case res' of
7        (EProgram (Just (SignalTerm e')) t'') ->
8          let tList = (Thread (numT+1) ctx ct input' a)
9          in EProgram (Just (SignalTerm e')) ([tList]++t'++t'')
10       (EProgram (Just e') t'') ->
11         let tList = (Thread (numT+1) ctx ct input' a)
12         in EProgram (Just (SignalTerm e')) ([tList]++t'++t'')
```

Fonte: Autoria própria.

A avaliação de uma expressão `lift` (Código-fonte 5.15) começa com a contagem de possíveis *threads* criadas na avaliação de seus termos recebidos reativamente, para tratar da possibilidade de uma expressão reativa ser usada como parâmetro de outra. Dessa maneira a contagem de *threads* se mantém consistente e a ordem se mantém correta para a avaliação de sinais posteriormente. Assim, é feita a avaliação da expressão `lift` utilizando seus valores iniciais, a partir da execução de uma expressão `InvokeClosure`.

O resultado obtido (`EProgram Term [Thread]`) contém o valor resultante da avaliação da expressão *lambda*, e a lista de *threads* geradas nesse passo de avaliação, que inclui a *thread* do próprio `lift`, e possíveis *threads* vindas de seus parâmetros.

Já no caso da avaliação de uma expressão `foldp` (Código-fonte 5.16), primeiro é feita a avaliação do único termo reativo da expressão para obter suas possíveis *threads* criadas. É então, feita a execução do termo `foldp`, utilizando um `InvokeClosure`, utilizando como parâmetros para a *closure* a variável `acc` e o termo reativo `t`. O resultado da avaliação da expressão `foldp` é o valor obtido, na forma de `SignalTerm`, e a lista de *threads* geradas, incluindo a *thread* do próprio `foldp`.

Por fim, a avaliação de uma expressão `async` (Código-fonte 5.17) segue o mesmo modelo de um `foldp`. As duas expressões precisam guardar o valor de seu resultado para uma próxima avaliação, mas com uma diferença importante em sua definição: um `foldp` sempre utiliza o resultado de sua avaliação anterior por meio da variável acumuladora `acc`, já no caso do `async`, seu resultado é guardado para criar a noção de assincronia entre os dados, que será vista na próxima etapa de avaliação. Por isso, para o caso desta primeira avaliação, só é feita a avaliação do termo `async` utilizando seus valores iniciais com o auxílio dos ambientes Γ e $Input$.

5.3.3.2 Avaliação de Sinal

A avaliação de sinais é oficialmente a última etapa de interpretação do programa. Neste ponto, já foi feita a avaliação inicial do termo, e foi obtido um resultado para o programa utilizando os valores iniciais das variáveis, restando somente as avaliações reativas. Uma avaliação reativa acontece quando um novo dado é recebido para uma variável reativa do programa. Novos valores, em teoria, são gerados pelo *Global Event Dispatcher* (GED), porém para simplificar a execução de um programa *AsyncRFJ*, todos os eventos já estão pré-carregados em uma lista, que está disponível neste ponto da avaliação.

O resultado da avaliação funcional, vista anterioremente, é um termo avaliado (na forma de `EProgram`) que contém o resultado inicial da avaliação e a lista de *threads* geradas pelo programa avaliado. O construtor `EProgram` faz parte da entrada para a função que inicia a avaliação de sinais, chamada `evalEvents`, em conjunto com a lista de eventos a serem avaliados utilizando as *threads* geradas. O Código-fonte 5.18 demonstra a função `evalEvents` por completo.

Código-fonte 5.18 – Código para a função *evalEvents*.

```
1 evalEvents :: (EProgram, [(String,Term)]) -> [[Status]]
2 evalEvents ((EProgram (Just e) t), (g:gs)) =
3   let thisT = (executeThreads t t g)
4       t' = Data.List.map (\(_,newT)->newT) thisT
```

```
5       s' = Data.List.map (\(newS,_)->newS) thisT
6   in [s']++(evalEvents ((EProgram (Just e) t'),gs))
7 evalEvents ((EProgram (Just e) t), []) = []
```

Fonte: Autoria própria.

A função `evalEvents` é definida de uma maneira recursiva, onde cada chamada da função resulta em n *threads* e uma lista contendo n `Status`, as *threads* são obtidas a partir da avaliação do primeiro evento da lista de eventos recebida, e cada `Status` deve ser intepretado como o resultado da avaliação de uma *thread*. Enquanto a lista de `Status` é concatenada ao resultado da função, as *threads* modificadas são usadas como parâmetro para a próxima chamada recursiva, dessa maneira, é possível manter a corretude dos termos executados dentro das *threads* (especificamente `foldp` e `async`). Vale notar que a noção de "*threads* modificadas" implica somente modificações em seu ambiente $Input$, pois modificações no ambiente Γ, tabela de classes e no termo avalido não são salvas, fazendo com que sempre seja utilizado seus valores originais vindos da avaliação funcional.

Para cada evento recebido pela função `evalEvents`, é feita uma chamada da função recursiva `executeThreads`. Onde os parâmetros usados são: a lista inteira de *threads*, que se mantém sem modificações para servir como auxílio; a lista de *threads* que será avaliada recursivamente, até que a lista esteja vazia; e o evento a ser avilado, na forma de uma tupla contendo a `String` correspondente ao nome da variável reativa recebendo um novo valor, e o novo valor na forma de um `Term`. A função `executeThreads` pode ser vista no Código-fonte 5.19.

Código-fonte 5.19 – Código para as funções *executeThreads* e *executeSingleThr*.

```
1  executeThreads :: [Thread] -> [Thread] -> (String,Term) -> [(Status,Thread)]
2  executeThreads allT (t@(Thread id ctx ct input term):ts) ged =
3    case (executeSingleThr allT t ged) of
4      (s,i) -> [(s,(Thread id ctx ct i term))]++(executeThreads allT ts ged)
5  executeThreads allT [] ged = []
6
7  executeSingleThr :: [Thread] -> Thread -> (String, Term) -> (Status,Input)
8  executeSingleThr allT (Thread id ctx ct input term) (v,t) =
9    let old = Data.Map.map
10         (\(t0,st,tr)->case st of Change v -> (t0,NoChange v, tr)
11                                  NoChange v -> (t0,NoChange v, tr)) input
12       input' = Data.Map.mapWithKey
13         (\idT (t0,(NoChange st),tr) ->
14           if (idT == v) then (t0,(Change t),tr) else (t0,(NoChange st),tr)) old
15   in evalSignal ctx ct input' term allT (v,t)
```

Fonte: Autoria própria.

Para cada execução da função `executeThreads` é feita a chamada da função `executeSingleTh` para avaliar a *thread* atual. O resultado da avaliação de uma única *thread* é: seu `Status`

(`Change` ou `NoChange` implicando se houve mudança ou não), e a *thread* resultante da avaliação. Os dois valores são retornados em uma tupla, e no final da execução da função `executeThreads`, o resultado é uma lista de tuplas (`Status`, `Thread`).

A definição da função `executeSingleThr`, como demonstrado no Código-fonte 5.19, é uma simples chamada da função `evalSignal`. Porém para garantir a execução correta, é feito um *reset* no ambiente de sinais $Input$, transformando o `Status` de todas as entradas em `NoChange`. Após o *reset* da lista $Input$, é feita uma modificação somente na variável correspondente ao evento sendo avaliado, transformando seu `Status` em `Change` e realizando a chamada de `evalSignal` com este novo $Input$.

Neste momento da execução do interpretador, já foi definda a ordem de execução dos eventos, dessa maneira, cada evento é aplicado a todas as *threads*, e cada *thread* utiliza valores sintaticamente corretos em seu ambiente $Input$. Só resta definir a execução do termo de cada *thread* e como é estruturado seu retorno.

O Código-fonte 5.20 demonstra um trecho da função `evalSignal`, que tem como objetivo avaliar recursivamente o termo de uma *thread*. Esse trecho em específico mostra como é feita a avaliação de valores, que se tornam o ponto de parada da recursão e definem se o retorno da *thread* é um valor `Change` ou `NoChange`, demonstrando se houve mudança nos sinais utilizados ou não.

Código-fonte 5.20 – Trecho do código para a função *evalSignal* para valores e variáveis.

```
1  evalSignal :: EnvG -> CT -> Input -> Term -> [Thread] -> (String,Term) ->
       (Status,Input)
2  evalSignal ctx ct input (Int e) v event = (NoChange (Int e),input)
3  evalSignal ctx ct input (BooleanLiteral e) v event = (NoChange (BooleanLiteral
       e),input)
4  evalSignal ctx ct input (Str e) v event = (NoChange (Str e),input)
5  evalSignal ctx ct input (Var e) v event =
6    case (Data.Map.lookup e input) of
7      Just (t,s,i) ->
8        if (i == -1) then (s,input)
9        else case (findThreadById v i) of (thr) -> executeSingleThr v thr event
10     Nothing -> case (Data.Map.lookup e ctx) of
11                  Just e' -> (NoChange e',input)
12                  Nothing -> error "Var not found"
```

Fonte: Autoria própria.

Vale notar que o retorno da função `evalSignal` é uma tupla contendo o `Status`, representando o valor resultado da avaliação do termo, e uma lista $Input$. Esta lista representa o ambiente de variáveis sinais após suas modificações, para ser usado na avaliação do próximo evento. Modificações no ambiente $Input$ só ocorrem na avaliação de `evalSignal` para o caso de um termo `foldp` ou `async`.

Outras avaliações importantes que ocorrem na função `evalSignal`, é o caso de

um termo `let` ou um termo `InvokeClosure`, demonstrado no Código-fonte 5.21. Para o caso de um termo `let`, primeiro é preciso avaliar o termo sendo atribuído à variável (termo `e1` na expressão `let var = e1 in e2`), que caso seja um valor `NoChange`, a variável é adicionada ao ambiente Γ e o próximo passo é avaliar `e2`. Caso contrário, o próximo passo também é avaliar `e2`, mas por ser um valor `Change`, obrigatóriamente o resultado de `e2` também é `Change`. Já para a avaliação de um termo `InvokeClosure`, os valores recebidos são adicionados ao ambiente Γ, sendo atribuídos aos aos seus respectivos parâmetros, e o próximo passo é avaliar o termo interno da *closure*.

Código-fonte 5.21 – Trecho do código para a função *evalSignal* para termos `let` e `InvokeClosure`.

```
1  evalSignal :: EnvG -> CT -> Input -> Term -> [Thread] -> (String,Term) ->
       (Status,Input)
2  evalSignal ctx ct input (Let var e1 e2) v event =
3    case (evalSignal ctx ct input e1 v event) of
4      (Change e,i) ->
5        let ctx' = (Data.Map.insert var e ctx)
6        in case (evalSignal ctx' ct input e2 v event) of
7             (Change e2',i') -> (Change e2',i')
8             (NoChange e2',i') -> (Change e2',i')
9      (NoChange e,i) ->
10       let ctx' = (Data.Map.insert var e ctx)
11       in (evalSignal ctx' ct input e2 v event)
12 evalSignal ctx ct input (InvokeClosure (ClosureDef par e2) e1) v event =
13   let p' = Data.List.zipWith (\(ty,var) (val) -> (var,val)) par e1
14       ctx' = Data.Map.union (Data.Map.fromList p') ctx
15   in evalSignal ctx' ct input e2 v event
```

Fonte: Autoria própria.

Também faz parte da função `evalSignal` avaliar os termos reativos das *threads*, para o caso de um termo `lift`, o Código-fonte 5.22 demonstra como sua avaliação é definida. O primeiro passo é avaliar cada um dos sinais do termo (lista de termos `e1`, na expressão `Lift p e e1`), aplicando o evento recebido. Com a utilização da função auxiliar `hasChanged`, é possível obter um valor booleano indicando se houve mudança ou não em qualquer um dos sinais. Utiliza-se a função auxiliar `getStatus` para gerar a nova lista de valores `newE1`, que será utilizada na execução do termo `lift`. Essa execução é feita a utilizando um termo `InvokeClosure`, pois sua avaliação aplica os valores recebidos aos parâmetros da função anônima e realiza sua execução. As funções auxiliares `hasChanged` e `getStatus` podem ser vistas no Código-fonte 5.23.

Código-fonte 5.22 – Trecho do código para a função *evalSignal* para termos `lift`.

```
1  evalSignal :: EnvG -> CT -> Input -> Term -> [Thread] -> (String,Term) ->
       (Status,Input)
```

```
2  evalSignal ctx ct input (Lift p e e1) v event = --R-ThreadLift
3    let e1'' = (Data.List.map (\x -> evalSignal ctx ct input x v event) e1)
4        e1' = (Data.List.map (\(x,i) -> x) e1'')
5        newE1 = (Data.List.map (\x -> getStatus x) e1')
6        changeBool = hasChanged e1'
7        res = (evalSignal ctx ct input
8                    (InvokeClosure (ClosureDef p e) newE1) v event)
9    in if changeBool then
10     case res of
11       (Change v',i) -> (Change v',input)
12       (NoChange v',i) -> (Change v',input)
13     else res
```

Fonte: Autoria própria.

Código-fonte 5.23 – Código para as funções auxiliares *hasChanged* e *getStatus*.

```
1  hasChanged :: [Status] -> Bool
2  hasChanged ((Change v):hs) = True || (hasChanged hs)
3  hasChanged ((NoChange v):hs) = False || (hasChanged hs)
4  hasChanged [] = False
5
6  getStatus :: Status -> Term
7  getStatus (Change v) = v
8  getStatus (NoChange v) = v
```

Fonte: Autoria própria.

Para a avaliação de um termo `foldp`, utiliza-se o trecho da função `evalSignal` visto no Código-fonte 5.24. Assim como o `lift`, é feita a avaliação do termo sinal, utilizando a função auxiliar `getStatus` para obter seu valor, e uma checagem para saber se houve mudança ou não (similar à função `hasChanged`). Porém, diferente do `lift`, uma expressão `foldp` contém somente uma variável sinal. É então, realizado um *lookup* na variável acumuladora `acc`, presente no ambiente $Input$, que é utilizada para a execução do `foldp` por meio de um termo `InvokeClosure`. Este novo resultado é guardado na variável `acc`, gerando um novo ambiente $Input$, para futuras avaliações. O restorno da avaliação do termo `foldp` é uma tupla contendo o novo valor resultado obtido, e o novo ambiente $Input$.

Código-fonte 5.24 – Trecho do código para a função *evalSignal* para termos `foldp`.

```
1  evalSignal :: EnvG -> CT -> Input -> Term -> [Thread] -> (String,Term) ->
       (Status,Input)
2  evalSignal ctx ct input (Foldp p e acc e1) v event = --R-ThreadFoldp
3    let e1'' = (evalSignal ctx ct input e1 v event)
4        e1' = (case e1'' of (x,i) -> x)
5        newE1 = (getStatus e1')
6        acc' = case (Data.Map.lookup "acc" input) of
7          Just (t0,(NoChange v),tr) -> v
```

```
8         changeBool = case e1' of (Change v) -> True
9                                  (NoChange v) -> False
10        res = (evalSignal ctx ct input
11          (InvokeClosure (ClosureDef p e) ([newE1]++[acc'])) v event)
12        input' = case res of
13          (s, i) -> let val = getStatus s
14            in Data.Map.mapWithKey (\idT (t0,st,tr) ->
15              if (idT=="acc") then (t0,(NoChange val),tr)
16              else (t0,st,tr)) input
17    in if changeBool then case res of
18                          (Change v',i) -> (Change v',input')
19                          (NoChange v',i) -> (Change v',input')
20        else res
```

Fonte: Autoria própria.

Já para o caso de um termo `async`, o Código-fonte 5.25 demonstra a avaliação feita pela função `evalSignal`. Diferente dos outros construtores reativos da linguagem, o `async` tem somente um termo em seu corpo, e não executa uma função anônima. A avaliação de uma expressão `async` consiste em "burlar" a ordem de avaliação do seu termo interno, de uma maneira em que a execução de uma expressão `async` supostamente não espera uma nova execução do termo em questão, e utiliza o valor de sua avaliação anterior. Dessa maneira, a expressão que chamou a avaliação do `async` não ficaria "presa" esprando o cálculo deste termo. Como a implementação deste trabalho não utiliza uma execução paralela real, e sim uma simulação da execução paralela de uma maneira sequencial, assume-se que o termo dentro do `async` nunca está pronto, e sempre é utilizado o valor do cálculo anterior.

Código-fonte 5.25 – Trecho do código para a função *evalSignal* para termos `async`.

```
1  evalSignal :: EnvG -> CT -> Input -> Term -> [Thread] -> (String,Term) ->
       (Status,Input)
2  evalSignal ctx ct input (Async e) v event = --R-ThreadAsync
3    let e1'' = (evalSignal ctx ct input e v event)
4        e1' = (case e1'' of (x,i) -> x)
5        newE1 = (getStatus e1')
6        async' = case (Data.Map.lookup "async" input) of
7          Just (t0,a',tr) -> getStatus a'
8          _ -> case (findThreadByTerm v (Async e)) of
9            (thr) -> case (executeSingleThread v thr event) of
10             (a',_) -> getStatus a'
11       input' = Data.Map.mapWithKey (\idT (t0,st,tr) ->
12                   if (idT=="async") then (t0,(NoChange newE1),tr)
13                   else (t0,st,tr)) input
14   in (NoChange async',input')
```

Fonte: Autoria própria.

Assim, a avaliação de uma expressão `async` na função `evalSignal`, faz o *lookup* no valor resultado da execução anterior no ambiente $Input$, e o utiliza como retorno. É também feito o cálculo do termo assíncrono e salvo nesta mesma variável, caso realmente tenha gerado um novo valor, dessa maneira a próxima execução terá acesso ao resultado mais recente da avaliação do termo.

Neste capítulo foram apresentadas as principais características do interpretador da linguagem orientada à objetos baseada no Featherweight Java, capaz de manipular dados reativos de uma maneira assíncrona, apresentando passo a passo os principais aspectos da sua implementação.

6 DISCUSSÃO E TRABALHOS RELACIONADOS

Programação Orientada a Objetos não disponibiliza mecanismos específicos para implementar comportamento reativo, resultando em algumas consequências. Primeiro, comportamento reativo normalmente é implementado usando o padrão *Observer*, cujas desvantagens já foram destacadas na literatura (COOPER; KRISHNAMURTHI, 2006)(MEYEROVICH et al., 2009)(MAIER; ODERSKY, 2012). Segundo, a funcionalidade de *update*, com as estratégias necessárias para atingir os requisitos de uma performance desejada, devem ser implementadas manualmente para cada aplicação. Tais otimizações, entretanto, introduzem bastante complexidade adicional, resultando na necessidade de um balanceamento entre complexidade e eficiência (SALVANESCHI; MEZINI, 2014).

Algumas abordagens tentam resolver diferentes aspectos desses problemas. *Event-Driven Programming* cria uma inversão de controle para permitir a modularização do código de *update* (RAJAN; LEAVENS, 2008)(EUGSTER; JAYARAM, 2009). Já *Aspect-Oriented Programming* permite a separação completa do *update*, e especifica nos aspectos os pontos onde seu uso é necessário (BODDEN; SHAIKH; HENDREN, 2008)(HANNEMANN; KICZALES, 2002). Essas duas abordagens se encaixam bem com objetos que podem sofrer mudanças, mas elas ainda carregam alguns dos problemas relacionados ao estilo de programação baseado em inversão de controle, semelhante aos problemas discutidos do padrão *Observer* (SALVANESCHI; MEZINI, 2014).

Sendo assim, abordagens declarativas e reativas se tornam uma boa escolha, pois fazem a automatização do processo de *update*, como visto em programação funcional reativa (ELLIOTT; HUDAK, 1997) e linguagens reativas como *FrTime* (COOPER; KRISHNAMURTHI, 2006), *Flapjax* (MEYEROVICH et al., 2009) e *Scala.React* (MAIER; ODERSKY, 2012). Isso gera uma formalização elegante com uma semântica denotacional em que comportamentos são modelados como funções de tempo para valores (ELLIOTT; HUDAK, 1997). A definição de como um valor mutável é computado, a partir de outros valores, é feita pelo desenvolvedor, e o *framework* garante que o valor computado seja automaticamente atualizado toda vez que a entrada sofre mudanças.

Linguagens reativas provêm abstrações para tratar de valores que mudam com o tempo. Para este trabalho, as abstrações utilizadas foram expressões `lift`, `foldp` e `async`. Para simplicidade de terminologia, uma expressão composta de uma ou mais abstrações reativas será chamada de *behavior*. Assim, um dos pontos importantes em trabalhos sobre linguagens reativas é, como *behaviors* devem ser incorporados ao modelo de orientação a objetos (SALVANESCHI; MEZINI, 2014).

Salvaneschi e Mezini (2014) provêm um estudo (*roadmap*) para superar as limitações das abordagens atuais do suporte de aplicações reativas, em um contexto de orientação a objeto. Em seu estudo sobre FRP, é argumentado que se *behaviors* se tornassem

parte da interface de um objeto, algumas características devem ser levadas em consideração: a possibilidade de definir um *behavior* como *public* ou *private*, a habilidade ou não de *behaviors* poderem ser reatribuídos, e o comportamento no caso de polimorfismo e herança.

O trabalho de Ignatoff, Cooper e Krishnamurthi (2006) explora o problema de adaptação de uma GUI orientada a objeto legado, utilizando como base os conceitos de *behaviors*, e eventos da programação reativa funcional. O principal objetivo da adaptação é o entendimento da direção em que as mudanças de estado fluem: da aplicação para o *toolkit*, do *toolkit* para a aplicação, ou nas duas direções. É usado *Scheme* para refinar a adaptação para sua forma mais abstrata, e implementar a interface completa para uma liguagem reativa funcional.

Já El-Zawawy (2015), apresenta um *framework* para operações assíncronas em um modelo de programação funcional orientado a objetos. Seu objetivo principal é apresentar um modelo para resolver problemas clássicos da técnica de programação assíncrona, como o bloqueamento do processo principal sendo executado. Uma semântica operacional também é incluída para garantir que a linguagem seja simples, porém poderosa o suficiente para expressar aplicações importantes de programação.

Como pode ser visto, existem vários aspectos do paradigma de programação reativa que podem ser estudados, em um contexto de uma linguagem orientada a objetos. Aspectos como, a definição estrutural de um objeto, onde construtores reativos fazem parte das características a serem declaradas, há também a questão do fluxo e propagação de dados, e sua interação com execução a de objetos, e por fim, o tratamento de assincronia durante a execução, pois dados reativos podem iniciar computações com tempos de execução diferentes. Em contrapartida, neste trabalho foi feita uma definição formal básica para trabalhar com programação reativa em um contexto orientado a objeto. A estrutura de um objeto não foi modificada, porém a definição formal permite que isso seja adicionado às ferramentas da linguagem, já o fluxo e propagação de dados implementado é baseado nas árvores de sintaxe abstratas geradas pelo analisador sintático. Por fim, a noção de uma execução assíncrona é introduzida, porém em uma versão simplificada, onde o paralelismo é simulado de uma maneira sequencial, para que o foco do trabalho se mantivesse na extensão sendo implementada.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este projeto apresentou uma extensão da linguagem Featherweight Java, com funcionalidades para manipular conceitos da computação reativa, simulando a manipulação de dados contínuos que tem como base a programação funcional, em um contexto orientado a objetos. Como contribuição desta dissertação podemos elencar a descrição de uma semântica formal simples para trabalhar com programas ou algoritmos considerando as características da programação reativa, através do paradigma de orientação a objetos. Além disso, foi demonstrada a implementação de um interpretador para o FJ com a extensão proposta, como uma maneira de verificar as definições da semântica operacional e do sistema de tipos.

Atualmente existem outras pesquisas que aplicam abstrações reativas em um contexto de orientação a objeto, porém suas abordagens não exploram muito a relação entre o paradigma de programação reativa e orientação a objeto, no sentido de, utilizar todos os recursos do modelo OO reativamente. Como resultado, uma investigação sistemática da integração entre os recursos do modelo de orientação a objeto e abstrações reativas ainda é necessária. Este trabalho fornece a formalização de uma linguagem que suporta construtores de programação reativa, utilizando um *core* mínimo bem estabelecido na literatura, assim, é possível servir com base para futuros estudos na integração entre o modelo de orientação a objeto e programação reativa.

Em relação aos objetivos deste trabalho, pode-se afirmar que, foi possível utilizar conceitos de linguagens funcionais, originados da programação reativa funcional, em um contexto orientado a objeto, e a semântica proposta foi embutida e implementada no FJ com sucesso, mantendo as características de segurança em se tratando dos tipos da linguagem Java original. A linguagem *Haskell* mostrou-se adequada como ferramenta para a implementação do interpretador, pois as definições expressas através da semântica operacional são facilmentes traduzidas para uma linguagem funcional.

Com a implementação uma extensão para uma linguagem amplamente aceita pela indústria de software, foi possível obter um melhor entendimento na relação entre os dois paradigmas estudados, de programação reativa e orientação a objeto. Porém, um dos principais obstáculos encontrados para a definição da linguagem proposta foi sua a complexidade, quantos detalhes devem ser omitidos sendo que, ao mesmo tempo, seja mantido o aspecto reativo da linguagem. Os principais exemplos disso são o número limitado de construtores reativos, e tipos de dados que podem ser utilizados reativamente.

7.1 TRABALHOS FUTUROS

Para o processo de avaliação do programa no interpretador implementado foi feita a utilização de *threads*, que insere a noção de execução assíncrona. Para este trabalho foi feita a escolha de utilizar uma simulação de `threads`, realizando uma execução síncrona onde os dados são gerenciados de uma maneira assíncrona. Essa escolha foi feita principalmente para manter o código simples e o foco do trabalho se mantivesse na parte reativa do programa. Sendo assim, este é um dos principais tópicos para um aperfeiçoamento da linguagem futuramente: a utilização de `threads` reais na implementação do interpretador.

Em relação à integração entre programação reativa com uma linguagem orientada a objetos, este trabalho tratou das situações mais básicas, onde um programa pode fazer uso dos dois paradigmas. O próximo passo seria aperfeiçoar a linguagem de maneira que os dois paradigmas se sobreponham, por exemplo, da mesma maneira que um objeto sempre pode criar e manipular variáveis comuns em atributos, incluir a possibilidade de uma variável especial, e sua manipulação, que trata de dados reativos é uma maneira de integrar os dois paradigmas. Neste ponto, é prciso argumentar se um campo reativo dentro do objeto deve ser simples e manipulável, como uma variável comum, ou complexo e estático, como uma definição de método, onde sua única manipulação é ser executado.

Utilizando outro ponto de vista na integração entre os paradigmas, trazendo orientação a objeto à programação reativa, seria a utilização de chamada de métodos como funções alvo em um construtor reativo. Assim, seria possível utilizar objetos comum, por exemplo um dicionário, tendo funções internas para tradução, em uma chamada reativa que traduz palavras recebidas do usuário. Para este trabalho, uma função anônima e uma chamada de método são sintaticamente diferentes, porém uma adaptação pode ser feita para os contrutores reativos.

REFERÊNCIAS BIBLIOGRÁFICAS

ARIOLA, Z. M.; FELLEISEN, M. The call-by-need lambda calculus. **Journal of functional programming**, Cambridge University Press, v. 7, n. 3, p. 265–301, 1997.

BAINOMUGISHA, E. et al. A survey on reactive programming. **ACM Computing Surveys (CSUR)**, ACM, v. 45, n. 4, p. 52, 2013.

BELLIA, M.; OCCHIUTO, M. E. Proving type safety for java simple closures. In: **19th Concurrency, Specification and Programming-CS&P'2010**. [S.l.: s.n.], 2010. v. 1, p. 61–72.

BLACKHEATH, S.; JONES, A. **Functional Reactive Programming**. [S.l.]: Manning Publications Company, 2016.

BODDEN, E.; SHAIKH, R.; HENDREN, L. Relational aspects as tracematches. In: ACM. **Proceedings of the 7th international conference on Aspect-oriented software development**. [S.l.], 2008. p. 84–95.

COOPER, G. H.; KRISHNAMURTHI, S. Embedding dynamic dataflow in a call-by-value language. In: SPRINGER. **European Symposium on Programming**. [S.l.], 2006. p. 294–308.

CZAPLICKI, E.; CHONG, S. Asynchronous functional reactive programming for guis. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2013. v. 48, n. 6, p. 411–422.

EL-ZAWAWY, M. A. A robust framework for asynchronous operations on a functional object-oriented model. In: IEEE. **Cloud Computing (ICCC), 2015 International Conference on**. [S.l.], 2015. p. 1–6.

ELLIOTT, C. **Specification for a Functional Reactive Programming language**. 2011. Disponível em: <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>.

ELLIOTT, C.; HUDAK, P. Functional reactive animation. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 1997. v. 32, n. 8, p. 263–273.

EUGSTER, P.; JAYARAM, K. Eventjava: An extension of java for event correlation. In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 2009. p. 570–594.

FEITOSA, S. da S. **Source-code for FJQuantum Interpreter**. [S.l.]: GitHub, 2016. <https://github.com/fjquantum/master>.

____. **QuickCheck Property-based testing for Featherweight Java**. [S.l.]: GitHub, 2018. <https://github.com/sfeitosa/fj-qc>.

FELLEISEN, M.; FRIEDMAN, D. P. **A little Java, a few patterns**. [S.l.]: MIT press, 1998.

FOSTER, N. et al. Frenetic: A network programming language. **ACM Sigplan Notices**, ACM, v. 46, n. 9, p. 279–291, 2011.

HANNEMANN, J.; KICZALES, G. Design pattern implementation in java and aspectj. In: ACM. **ACM Sigplan Notices**. [S.l.], 2002. v. 37, n. 11, p. 161–173.

HUDAK, P. et al. Arrows, robots, and functional reactive programming. In: SPRINGER. **International School on Advanced Functional Programming**. [S.l.], 2002. p. 159–187.

IGARASHI, A.; PIERCE, B. C.; WADLER, P. Featherweight java: a minimal core calculus for java and gj. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 23, n. 3, p. 396–450, 2001.

IGNATOFF, D.; COOPER, G. H.; KRISHNAMURTHI, S. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In: SPRINGER. **International Symposium on Functional and Logic Programming**. [S.l.], 2006. p. 259–276.

KRISHNASWAMI, N. R.; BENTON, N.; HOFFMANN, J. Higher-order functional reactive programming in bounded space. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2012. v. 47, n. 1, p. 45–58.

LIU, H.; CHENG, E.; HUDAK, P. Causal commutative arrows and their optimization. In: ACM. **ACM Sigplan Notices**. [S.l.], 2009. v. 44, n. 9, p. 35–46.

LIU, H.; HUDAK, P. Plugging a space leak with an arrow. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 193, p. 29–45, 2007.

MAIER, I.; ODERSKY, M. **Deprecating the Observer Pattern with Scala. React**. [S.l.], 2012.

MEYEROVICH, L. A. et al. Flapjax: a programming language for ajax applications. In: ACM. **ACM SIGPLAN Notices**. [S.l.], 2009. v. 44, n. 10, p. 1–20.

NEWTON, R.; MORRISETT, G.; WELSH, M. The regiment macroprogramming system. In: IEEE. **Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on**. [S.l.], 2007. p. 489–498.

NILSSON, H. Dynamic optimization for functional reactive programming using generalized algebraic data types. In: CITESEER. **ICFP**. [S.l.], 2005. v. 5, p. 54–65.

NILSSON, H.; COURTNEY, A.; PETERSON, J. Functional reactive programming, continued. In: ACM. **Proceedings of the 2002 ACM SIGPLAN workshop on Haskell**. [S.l.], 2002. p. 51–64.

PETERSON, J.; TRIFONOV, V.; SERJANTOV, A. Parallel functional reactive programming. In: SPRINGER. **International Symposium on Practical Aspects of Declarative Languages**. [S.l.], 2000. p. 16–31.

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

RAJAN, H.; LEAVENS, G. T. Ptolemy: A language with quantified, typed events. In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 2008. p. 155–179.

SALVANESCHI, G.; MEZINI, M. Towards reactive programming for object-oriented applications. In: **Transactions on Aspect-Oriented Software Development XI**. [S.l.]: Springer, 2014. p. 227–261.

SCULTHORPE, N.; NILSSON, H. Safe functional reactive programming through dependent types. **ACM Sigplan Notices**, ACM, v. 44, n. 9, p. 23–34, 2009.

STALTZ, A. **The introduction to Reactive Programming you've been missing**. [S.l.]: GitHub, 2014. <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>.

TRAN, T. M. T.; STEFFEN, M. Safe commits for transactional featherweight java. In: SPRINGER. **International Conference on Integrated Formal Methods**. [S.l.], 2010. p. 290–304.

WAN, Z.; TAHA, W.; HUDAK, P. Real-time frp. **ACM SIGPLAN Notices**, ACM, v. 36, n. 10, p. 146–156, 2001.

____. Event-driven frp. In: SPRINGER. **International Symposium on Practical Aspects of Declarative Languages**. [S.l.], 2002. p. 155–172.

WESTERSTÅHL, D. On mathematical proofs of the vacuity of compositionality. **Linguistics and Philosophy**, Springer, v. 21, p. 635–643, 1998.

XU, D. N.; KHOO, S.-C. Compiling real time functional reactive programming. In: ACM. **Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation**. [S.l.], 2002. p. 83–93.

APÊNDICE A – FORMALIZAÇÃO DA LINGUAGEM PROPOSTA

A.1 SINTAXE

Definição de Tipos

T ::= T_{FJ} | T_P | T_F | T_S

T_{FJ} ::= class

T_P ::= boolean | string | int

T_F ::= T -> T_{FJ} | T -> T_P | T -> T_F

T_S ::= signal T | T -> T_S

Declarações de classes

CL ::= class C extends C { $\overline{C}$ $\overline{f}$; K $\overline{M}$ }

Declarações de construtores

K ::= C($\overline{C}$ $\overline{f}$) { super($\overline{f}$); this.$\overline{f}$=$\overline{f}$; }

Declarações de métodos

M ::= C m($\overline{C}$ $\overline{f}$) { return t; }

Termos

t ::= x
- | t.f
- | t.m($\overline{t}$)
- | new C($\overline{t}$)
- | true
- | false
- | int
- | string
- | t + t
- | t - t
- | t / t
- | t * t
- | if (t) { t } else { t }
- | let x = t in t
- | ($\overline{T}$ $\overline{x}$) -> t
- | t.invoke($\overline{t}$)

Termos (Cont.)

- | signal t
- | lift t t_1 ... t_n
- | foldp t_1 t_2 t_3
- | async t

Valores

v ::= new C($\overline{v}$)
- | true
- | false
- | int
- | string
- | signal v
- | ($\overline{T}$ $\overline{x}$) -> t

A.2 DEFINIÇÕES AUXILIARES

Regras de subtipo para as classes.

$$\frac{}{\text{C} <: \text{C}}$$

$$\frac{\text{C} <: \text{D} \quad \text{D} <: \text{E}}{\text{C} <: \text{E}}$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\ldots\}}{\text{C} <: \text{D}}$$

Obtenção de atributos de uma classe.

$$\textit{fields}(\text{Object}) = \bullet$$

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\} \quad \textit{fields}(\text{D}) = \overline{\text{D}}\ \overline{\text{g}}}{\textit{fields}(\text{C}) = \overline{\text{D}}\ \overline{\text{g}}, \overline{\text{C}}\ \overline{\text{f}}}$$

Obtenção do corpo de um método.

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\} \quad \text{B m}(\overline{\text{B}}\ \overline{\text{x}})\{\ \text{return e;}\ \} \in \overline{\text{M}}}{\textit{mbody}(\text{m, C}) = \overline{\text{x}}.\text{e}}$$

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\}\ \text{m} \notin \overline{\text{M}}}{\textit{mbody}(\text{m, C}) = \textit{mbody}(\text{m, D})}$$

Obtenção do tipo de um método.

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\} \quad \text{B m}(\overline{\text{B}}\ \overline{\text{x}})\{\ \text{return e;}\ \} \in \overline{\text{M}}}{\textit{mtype}(\text{m, C}) = \overline{\text{B}} \rightarrow \text{B}}$$

$$\frac{\text{class C extends D } \{\overline{\text{C}}\ \overline{\text{f}};\ \text{K}\ \overline{\text{M}}\}\ \text{m} \notin \overline{\text{M}}}{\textit{mtype}(\text{m, C}) = \textit{mtype}(\text{m, D})}$$

A.3 SISTEMA DE TIPOS

A.3.1 Formação da Tabela de Classes

Regra para classes bem definidas:

$$\frac{K = C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\ \{\ super(\overline{g});\ this.\overline{f} = \overline{f}\ \} \qquad \textit{fields}(D) = \overline{D}\ \overline{g} \qquad \overline{M}\ \text{OK em C}}{\text{class C extends D } \{\ \overline{C}\ \overline{f};\ K\ \overline{M}\ \}\ \text{OK}}$$

Regra para métodos bem definidos:

$$\frac{\overline{x} : \overline{C},\ this : C \vdash t_0 : E_0 \qquad E_0 <: C_0 \qquad \textit{CT}(C) = \text{class C extends D } \{\ \ldots\ \} \qquad \textit{override}(m, D, \overline{C} \rightarrow C_0)}{C_0\ m(\overline{C}\ \overline{x})\ \{\ \text{return } t_0;\ \}\ \text{OK em C}}$$

A.3.2 Tipagem dos Termos

Regras de tipo para os termos originais FJ:

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \textit{(Variáveis)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad fields(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i : C_i} \quad \textit{(Acesso a Atributos)}$$

$$\frac{\Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0) = \overline{D} \rightarrow C \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.m(\overline{e}) : C} \quad \textit{(Invocação de Métodos)}$$

$$\frac{fields(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \text{new C}(\overline{e}) : C} \quad \textit{(Criação de Objetos)}$$

Regras de tipo para termos booleanos e condicionais:

$$\frac{}{\text{true} : \text{boolean}} \qquad \frac{}{\text{false} : \text{boolean}} \quad \textit{(Booleanos)}$$

$$\frac{t_1 : \text{boolean} \quad t_2 : T \quad t_3 : T}{\text{if } (t_1) \{ t_2 \} \text{ else } \{ t_3 \} : T} \quad \textit{(Condicionais)}$$

Regras de tipo para números inteiros e operações matemáticas:

$$\frac{}{\Gamma \vdash \text{int} : \text{Integer}} \quad \textit{(Inteiros)}$$

$$\frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 + t_2 : \text{int}} \qquad \frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 - t_2 : \text{int}}$$

$$\frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 * t_2 : \text{int}} \qquad \frac{\Gamma \vdash t_1, t_2 : \text{int}}{\Gamma \vdash t_1 / t_2 : \text{int}} \quad \textit{(Operadores)}$$

Regras de tipo para o termo let:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : T_2} \quad \textit{(Let)}$$

Regras de tipo para closures:

$$\frac{\overline{T} \text{ OK} \quad \Gamma, \overline{x} : \overline{T} \vdash t : T}{\Gamma \vdash (\overline{T}\ \overline{x}) \to t : (\overline{T} \to T)} \quad \textit{(Closure bem definida)}$$

$$\frac{\Gamma \vdash t : (\overline{T} \to T) \quad \Gamma \vdash \bar{t} : \overline{S} \quad \overline{S} <: \overline{T}}{\Gamma \vdash t.\text{invoke}(\bar{t}) : T} \quad \textit{(Invocação de closure)}$$

Regras de tipo para construtores reativos:

$$\frac{I\ (i) = \text{signal } \tau}{I \vdash i : \text{signal } \tau} \quad \textit{(Sinal)}$$

$$\frac{\Gamma \vdash lam : \tau_1 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash t_i : \text{signal } \tau_i \quad \forall i \in 1..n}{\Gamma \vdash \text{lift } lam\ t_1 \ldots t_n : \text{signal } \tau} \quad \textit{(Lift)}$$

$$\frac{\Gamma \vdash lam : \tau \to \tau' \to \tau' \quad \Gamma \vdash a : \tau' \quad \Gamma \vdash t : \text{signal } \tau}{\Gamma \vdash \text{foldp } lam\ a\ t : \text{signal } \tau'} \quad \textit{(Foldp)}$$

$$\frac{\Gamma \vdash t : \text{signal } \tau}{\Gamma \vdash async\ t : \text{signal } \tau} \quad \textit{(Async)}$$

A.4 AVALIAÇÃO SEMÂNTICA

A.4.1 Regras de Redução

Regras de avaliação para os termos originais FJ:

$$\frac{\textit{fields}(\mathsf{C}) = \overline{\mathsf{C}}\ \overline{\mathsf{f}}}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{f}_i \rightarrow \mathsf{v}_i} \quad \textit{(Acesso a Atributos)}$$

$$\frac{\textit{mbody}(\mathsf{m}, \mathsf{C}) = \overline{\mathsf{x}}.\mathsf{e}_0}{(\mathsf{new\ C}(\overline{\mathsf{v}})).\mathsf{m}(\overline{\mathsf{d}}) \rightarrow [\overline{\mathsf{d}} \ / \ \overline{\mathsf{x}}, \mathsf{new\ C}(\overline{\mathsf{v}}) \ / \ \mathsf{this}]\mathsf{e}_0} \quad \textit{(Invocação de Métodos)}$$

Regras de avaliação para termos condicionais:

$$\frac{}{\mathsf{if\ (true)\ \{\ t_2\ \}\ else\ \{\ t_3\ \} \rightarrow t_2} \quad \textit{(Condição True)}$$

$$\frac{}{\mathsf{if\ (false)\ \{\ t_2\ \}\ else\ \{\ t_3\ \} \rightarrow t_3} \quad \textit{(Condição False)}$$

Regras de avaliação para o operador let:

$$\frac{}{\mathsf{let\ x = v_1\ in\ t_2 \rightarrow [x \mapsto v_1]\ t_2} \quad \textit{(Operador Let)}$$

Regras de avaliação para closures:

$$\frac{}{((\overline{\mathsf{T}}\ \overline{\mathsf{x}}) \rightarrow \mathsf{T})\ \mathsf{t.invoke}(\overline{\mathsf{d}}) \longrightarrow [\overline{\mathsf{d}} \mapsto \overline{\mathsf{x}}]\ \mathsf{t}} \quad \textit{(Invocação de closures)}$$

Regras de avaliação para construtores reativos:

$$\frac{Input(s_i) = v_i\ \forall i \in (1..n)}{\Gamma, Input \vdash \mathsf{lift}\ lam\ s_1 \ldots s_n \longrightarrow lam\ .\ \mathsf{invoke}\ (v_1 \ldots v_n)} \quad \textit{(Lift)}$$

$$\frac{Input(s) = v}{\Gamma, Input \vdash \mathsf{foldp}\ lam\ acc\ s \longrightarrow lam\ .\ \mathsf{invoke}\ (acc\ v)} \quad \textit{(Foldp)}$$

$$\frac{Input(s) = v}{\Gamma, Input \vdash \mathsf{async}\ s \longrightarrow v} \quad \textit{(Async)}$$

Regras de avaliação de threads:

$$\frac{\forall i \in (0..n)\ \overline{t_i\langle e\rangle}\ g_i \longrightarrow \overline{t_{i+1}\langle e\rangle}\ \overline{\mathsf{Status}\ \mathsf{e}_i}}{\overline{t_0\langle e\rangle}\ g_0, \ldots, g_n \longrightarrow \overline{t_{n+1}\langle e\rangle}\ \overline{\mathsf{Status}\ \mathsf{e}_0}, \ldots, \overline{\mathsf{Status}\ \mathsf{e}_n}} \quad \textit{(GED)}$$

$$\frac{\forall i \in (0..n)\ Input(s_i) = \mathsf{NoChange}\ ()}{\Gamma, Input \vdash t\langle \mathsf{lift}\ lam\ s_1 \ldots s_n\rangle \longrightarrow \mathsf{NoChange}\ ()} \quad \textit{(Lift NoChange)}$$

$$\frac{\begin{array}{c}\exists i \in (0..n)\ Input(s_i) = \mathsf{Change}\ e \\ \Gamma, Input \vdash lam.\ \mathsf{invoke}\ (v_1 \ldots v_n) \rightarrow Input' \vdash \mathsf{Change}\ e'\end{array}}{\Gamma, Input \vdash t\langle \mathsf{lift}\ lam\ s_1 \ldots s_n\rangle \longrightarrow Input' \vdash \mathsf{Change}\ e'} \quad \textit{(Lift Change)}$$

$$\frac{Input(s) = \mathsf{NoChange}\ ()}{\Gamma, Input \vdash t\langle \mathsf{foldp}\ lam\ acc\ s\rangle \longrightarrow \mathsf{NoChange}\ ()} \quad \textit{(Foldp NoChange)}$$

$$\frac{\begin{array}{c}Input(s) = \mathsf{Change}\ e \\ \Gamma, Input \vdash lam.\ \mathsf{invoke}\ (acc\ v) \rightarrow Input' \vdash \mathsf{Change}\ e'\end{array}}{\Gamma, Input \vdash t\langle \mathsf{foldp}\ lam\ acc\ s\rangle \longrightarrow Input' \vdash \mathsf{Change}\ e'} \quad \textit{(Foldp Change)}$$

$$\frac{Input(s) = \mathsf{NoChange}\ ()}{\Gamma, Input \vdash t\langle \mathsf{async}\ s\rangle \longrightarrow \mathsf{NoChange}\ ()} \quad \textit{(Async NoChange)}$$

$$\frac{Input(s) = \mathsf{Change}\ e}{\Gamma, Input \vdash t\langle \mathsf{async}\ s\rangle \longrightarrow \mathsf{Change}\ e} \quad \textit{(Async Change)}$$

A.4.2 Regras de Congruência

Regras de congruência para os termos originais FJ:

$$\frac{t_0 \to t_0'}{t_0.f \to t_0'.f} \quad \textit{(Acesso a atributos)}$$

$$\frac{t_0 \to t_0'}{t_0.m(\bar{t}) \to t_0'.m(\bar{t})} \quad \textit{(Invocação de métodos)}$$

$$\frac{t_i \to t_i'}{v_0.m(\overline{v}, t_i, \bar{t}) \to v_0.m(\overline{v}, t_i', \bar{t})} \quad \textit{(Parâmetros dos métodos)}$$

$$\frac{t_i \to t_i'}{\text{new } C(\overline{v}, t_i, \bar{t}) \to \text{new } C(\overline{v}, t_i', \bar{t})} \quad \textit{(Criação de objetos)}$$

Regras de congruência para termos condicionais:

$$\frac{t_1 \to t_1'}{\text{if } (t_1) \{ t_2 \} \text{ else } \{ t_3 \} \to t_2 \to \text{if } (t_1') \{ t_2 \} \text{ else } \{ t_3 \} \to t_2} \quad \textit{(Condicionais)}$$

Regras de congruência para operadores matemáticos:

$$\frac{t_1 \to t_1'}{t_1 + t_2 \to t_1' + t_2} \qquad \frac{t_2 \to t_2'}{v_1 + t_2 \to v_1 + t_2'} \quad \textit{(Operador soma)}$$

$$\frac{t_1 \to t_1'}{t_1 - t_2 \to t_1' - t_2} \qquad \frac{t_2 \to t_2'}{v_1 - t_2 \to v_1 - t_2'} \quad \textit{(Operador subtração)}$$

$$\frac{t_1 \to t_1'}{t_1 * t_2 \to t_1' * t_2} \qquad \frac{t_2 \to t_2'}{v_1 * t_2 \to v_1 * t_2'} \quad \textit{(Operador multiplicação)}$$

$$\frac{t_1 \to t_1'}{t_1 / t_2 \to t_1' / t_2} \qquad \frac{t_2 \to t_2'}{v_1 / t_2 \to v_1 / t_2'} \quad \textit{(Operador divisão)}$$

Regras de congruência para o operador let:

$$\frac{t_1 \to t_1'}{\text{let } x = t_1 \text{ in } t_2 \to \text{let } x = t_1' \text{ in } t_2} \quad \textit{(Operador Let)}$$

Regras de congruência para closures:

$$\frac{t \to t'}{t.\text{invoke}(\bar{t}) \longrightarrow t'.\text{invoke}(\bar{t})} \quad \textit{(Invocação de closures)}$$

$$\frac{t \to t'}{t.\text{invoke}(\ldots, t, \ldots) \longrightarrow t.\text{invoke}(\ldots, t', \ldots)} \quad \textit{(Parâmetros das closures)}$$

Regras de congruência para construtores reativos:

$$\frac{x \notin fv(F[\cdot]) \quad Input' = \ldots, add(x,s)}{\Gamma, Input \vdash E[\mathsf{let}\ x = s\ \mathsf{in}\ u] \longrightarrow Input' \vdash \mathsf{let}\ x = s\ \mathsf{in}\ E[u]} \quad \textit{(Let Sinal)}$$

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{lift}\ lam\ e_1 \ldots e_n\rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{lift}\ lam\ e_1 \ldots e_n)\ \mathsf{T} \longrightarrow (\mathsf{lift}\ lam\ e_1 \ldots e_n)\ \mathsf{T'}} \quad \textit{(Lift Thread)}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{lift}\ lam\ s_1 \ldots e \ldots e_n)\ \mathsf{T} \longrightarrow (\mathsf{lift}\ lam\ s_1 \ldots e' \ldots e_n)\ \mathsf{T'}} \quad \textit{(Lift Sinais)}$$

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{foldp}\ lam\ acc\ e\rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T} \longrightarrow (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T'}} \quad \textit{(Foldp Thread)}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{foldp}\ lam\ acc\ e)\ \mathsf{T} \longrightarrow (\mathsf{foldp}\ lam\ acc\ e')\ \mathsf{T'}} \quad \textit{(Foldp Sinais)}$$

$$\frac{\mathsf{T'} = \mathsf{T} \parallel spawn\ t\langle \mathsf{async}\ e\rangle : \Gamma, Input}{\Gamma, Input \vdash (\mathsf{async}\ e)\ \mathsf{T} \longrightarrow (\mathsf{async}\ e)\ \mathsf{T'}} \quad \textit{(Async Thread)}$$

$$\frac{\Gamma, Input \vdash \mathsf{e} \rightarrow \mathsf{e'}\ \mathsf{t'} \quad \mathsf{T'} = \mathsf{T} \parallel \mathsf{t'}}{\Gamma, Input \vdash (\mathsf{async}\ e)\ \mathsf{T} \longrightarrow (\mathsf{async}\ e')\ \mathsf{T'}} \quad \textit{(Async Sinais)}$$

APÊNDICE B – GRAMÁTICA BNF E CONSTRUTORES ASTS

B.1 GRAMÁTICA BNF PARA A LINGUAGEM PROPOSTA

```
1  -- Codigos do programa
2  Program : ClassList InputList EventList Term ';'  { Program $1 $2 $3 $4 }
3
4  -- Lista de Inputs
5  InputList : '{' InputDefList '}'                  { $2 }
6
7  InputDefList : '(' name ',' Term ')'              { [($2,$4,(NoChange $4))] }
8          | InputDefList ',' '(' name ',' Term ')'  { ($4,$6,(NoChange $6)):$1 }
9
10 -- Lista de Events
11 EventList : '{' EventDefList '}'                  { $2 }
12
13 EventDefList : '(' name ',' Term ')'              { [($2,$4)] }
14         | EventDefList ',' '(' name ',' Term ')'  { ($4,$6):$1 }
15
16 -- Lista de classes
17 ClassList : ClassDef                              { [$1] }
18         | ClassList ClassDef                      { $2 : $1 }
19
20 -- Definicao de classe
21 ClassDef : class ClassName extends ClassName
22         '{' AttrList ConstrDef MethodList '}'  { ($2,ClassDef $2 $4 $6 $7 $8) }
23
24 -- Definicao de construtor
25 ConstrDef : ClassName '(' ParamList ')' '{' super '(' ArgList ')' ';'
       AttrAssignList '}'        { ConstrDef $1 $3 $8 $11 }
26
27 -- Lista de implementacao de metodos
28 MethodList : {- empty -}                           { [] }
29         | MethodList MethodDef                    { $2 : $1 }
30
31 -- Definicao de metodos
32 MethodDef : TypeDef name '(' ParamList ')' '{' return Term ';' '}'
                    { MethodDef $1 $2 $4 $8 }
33
34 -- Lista de declaracao de parametros para construtores e definicao de funcoes
35 -- Parametros Formais
36 ParamList : {- empty -}                            { [] }
37         | IdentStmt                               { [$1] }
```

```
38          | ParamList ',' IdentStmt                 { $3 : $1 }
39
40  -- Lista de argumentos utilizados para passar informacoes nas chamadas de
        funcoes
41  -- Parametros Atuais
42  ArgList : {- empty -}                             { [] }
43          | name                                    { [$1] }
44          | ArgList ',' name                        { $3 : $1 }
45
46  -- Lista de atributos de classe (declaracao)
47  AttrList : {- empty -}                            { [] }
48          | AttrList IdentStmt ';'                  { $2 : $1 }
49
50  -- Lista de atribuicao interna de atributos (utilizada no construtor da classe)
51  AttrAssignList : {- empty -}                      { [] }
52          | AttrAssign                              { [$1] }
53          | AttrAssignList AttrAssign               { $2 : $1 }
54
55  -- Atribuicao interna de atributos
56  AttrAssign : this '.' name '=' name ';'           { ($3,$5) }
57
58  -- Definicao de atributos ou parametros
59  IdentStmt : TypeDef name                          { ($1,$2) }
60
61  -- Definicao de tipos
62  TypeDef : boolean                                 { TypeBool }
63          | string                                  { TypeString }
64          | ClassName                               { TypeClass $1 }
65          | '(' TypeList "->" TypeDef ')'           { TypeClosure $4 $2 }
66          | '{' TypeList '}'                        { TypeTuple $2 }
67          | int                                     { TypeInt }
68
69  -- Lista de tipos utilizados para as Closures
70  TypeList : {- empty -}                            { [] }
71          | TypeDef                                 { [$1] }
72          | TypeList ',' TypeDef                    { $3 : $1 }
73
74  -- Nomes de classes
75  ClassName : Object                                { "Object" }
76          | name                                    { $1 }
77
78  -- Lista de termos
79  TermList : {- empty -}                            { [] }
80          | Term                                    { [$1] }
81          | TermList ',' Term                       { $3 : $1 }
82
83  -- Termos (utilizados no corpo dos metodos)
```

```
84  Term : BooleanLiteral                                   { BooleanLiteral $1 }
85         | name                                           { Var $1 }
86         | '"' name '"'                                   { Str $2 }
87         | this '.' name                                  { ThisAccessAttr $3 }
88         | this '.' name '(' TermList ')'                 { ThisAccessMeth $3 $5 }
89         | Term '.' name                                  { AttrAccess $1 $3 }
90         | Term '.' name '(' TermList ')'                 { MethodAccess $1 $3 $5 }
91         | new ClassName '(' TermList ')'                 { CreateObject $2 $4 }
92         | '(' ClassName ')' Term                         { Cast $2 $4 }
93         | if '(' Term ')' '{' Term '}' else '{' Term '}'
94                                                          { If $3 $6 $10 }
95         | let name '=' Term in Term                      { Let $2 $4 $6 }
96         | '(' ParamList ')' "->" Term                    { ClosureDef $2 $5 }
97         | '(' Term ')' '.' invoke '(' TermList ')'{ InvokeClosure $2 $7 }
98         | '{' TermList '}'                               { Tuple $2 }
99         | Term '.' number                                { TupleAccess $1 $3 }
100        | Term "==" Term                                 { RelEquals $1 $3 }
101        | number                                         { Int $1 }
102        | Term '*' Term                                  { Times $1 $3 }
103        | Term '/' Term                                  { Divide $1 $3 }
104        | Term '+' Term                                  { Plus $1 $3 }
105        | Term '-' Term                                  { Minus $1 $3 }
106        | lift '(' '(' ParamList ')' "->" Term ')' '(' TermList ')'
107                                                         { Lift $4 $7 $10 }
108        | foldp '(' '(' ParamList ')' "->" Term ')' Term '(' Term ')'
109                                                         { Foldp $4 $7 $9 $11 }
110        | async Term                                     { Async $2 }
111
112 -- Booleanos
113 BooleanLiteral : true                                   { BLTrue }
114        | false                                          { BLFalse }
```

B.2 CONSTRUTORES DAS ASTS

```
1  data Program = Program [(String,ClassDef)] [(String, Term, Status)] [(String,
       Term)] Term
2          deriving (Show, Eq)
3
4  data ClassDef = ClassDef String String [(Type,String)] ConstrDef [MethodDef]
5          deriving (Show, Eq)
6
7  data ConstrDef = ConstrDef String [(Type,String)] [String] [(String,String)]
8          deriving (Show, Eq)
9
10 data MethodDef = MethodDef Type String [(Type,String)] Term
```

```
11                          deriving (Show, Eq)
12
13  data Term = EmptyTerm
14          | BooleanLiteral BooleanLiteral
15          | Int Int
16          | Var String
17          | Str String
18          | ThisAccessAttr String
19          | ThisAccessMeth String [Term]
20          | AttrAccess Term String
21          | MethodAccess Term String [Term]
22          | CreateObject String [Term]
23          | Cast String Term
24          | If Term Term Term
25          | Let String Term Term
26          | ClosureDef [(Type,String)] Term
27          | InvokeClosure Term [Term]
28          | Tuple [Term]
29          | TupleAccess Term Int
30          | RelEquals Term Term
31          | Times Term Term
32          | Divide Term Term
33          | Minus Term Term
34          | Plus Term Term
35          | Lift [(Type,String)] Term [Term]
36          | Foldp [(Type,String)] Term Term Term
37          | Async Term
38          | SignalTerm Term
39          deriving (Show,Eq)
40
41  data TermNL = VarNL Int
42          | ClosureDefNL [Type] TermNL
43          | InvokeClosureNL TermNL [TermNL]
44          deriving (Show, Eq)
45
46  data BooleanLiteral     = BLTrue
47          | BLFalse
48          deriving (Show, Eq)
49  data Status = Change Term
50          | NoChange Term
51          deriving (Show, Eq)
52
53  data Type = TypeBool
54          | TypeString
55          | TypeClass String
56          | TypeClosure Type [Type]
57          | TypeTuple [Type]
```

```
58          | TypeInt
59          | SignalType Type
60          deriving (Show, Eq)
61
62  type Input = Map String (Term,Status,Int)
63  type EnvG = Map String Term
64  type Env = Map String Type
65  type CT = Map String ClassDef
```