Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Formalização em Redex de uma Abordagem Baseada em Tipo Para Terminação de PEG

Regina Sarah Monferrari Amorim de Paula

JUIZ DE FORA
DEZEMBRO, 2023

Formalização em Redex de uma Abordagem Baseada em Tipo Para Terminação de PEG

Regina Sarah Monferrari Amorim de Paula

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Leonardo V. S. Reis

Coorientador: Elton M. Cardoso

JUIZ DE FORA

DEZEMBRO, 2023

Ao meu amor e aos amigos.

Aos pais e avós, pelo apoio e sustento.

Resumo

O presente trabalho lida com o problema de terminação de PEGs usando uma abordagem baseada em sistema de tipos. Para alcançar esse objetivo, foi formalizado um sistema de inferência de tipos em PLT Redex usando semântica de reescrita.

Palavras-chave: PEG, *parsing*, sistema de tipos, inferência, Redex.

Abstract

The present work deals with the PEG termination problem using a type system-based approach. To achieve this goal, a type inference system was formalized in PLT Redex using rewriting semantics.

Keywords: PEG, parsing, type system, inference, Redex.

Agradecimentos

À minha família e amigos, pelo apoio, sustento e esperança que algum dia daria certo.

Ao Willian, que mesmo sem entender nada do que está escrito (e com razão) me apoiou, me encorajou e em muitos momentos, não me deixou desistir.

Aos professores Leonardo, Elton e Rodrigo pela orientação, suporte, paciência e didática, sem a qual este trabalho não se realizaria.

Ao Daniel, pela parceria e apoio durante todos esses anos trabalhando juntos.

Aos professores do Departamento de Ciência da Computação e de Matemática pelos seus ensinamentos e aos funcionários do curso, que durante esses anos, contribuíram de algum modo para o nosso enriquecimento pessoal e profissional.

"Nada contribui mais para tranquilizar a mente como um propósito firme – um ponto no qual a alma pode fixar seu olhar intelectual.".

Mary Shelley (Frankenstein)

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Abreviações

DCC	Departamento de Ciência da Computução
UFJF	Universidade Federal de Juiz de Fora
PEG	*Parsing Expression Grammar*
CFG	Gramáticas Livres de Contexto
RE	Expressões Regulares

1 Introdução

Na Ciência da Computação, linguagens formais são conjuntos abstratos de sequências de símbolos de acordo com regras definidas. Elas desempenham um papel fundamental na teoria da computação e na construção de autômatos para processar e reconhecer linguagens.

Baseando o entendimento de linguagem como a definição dada acima, construir uma linguagem de programação deve estar pautado nesses mesmos pilares: símbolos e regras. Todo sistema de computação tem como uma das etapas o reconhecimento de uma linguagem formal. A linguagem precisa ser interpretada e analisada, isto é, é preciso verificar se uma sequência de símbolos está em conformidade com um determinado conjunto de regras.

Qualquer informação inserida em uma máquina requer validação e análise, a fim de ser compatível com a linguagem de comando interpretada pela referida máquina. Um exemplo deste cenário é uma aplicação Web que necessita de um *parser*, uma análise, antes das páginas da aplicação serem exibidas.

O presente estudo trata sobre *Parsing Expression Grammar* (PEG) que é um formalismo para descrever reconhecedores de linguagens. Para tanto, é preciso definir um alfabeto, um conjunto de não terminais que farão parte da gramática, um conjunto de regras e um símbolo inicial de partida.

Uma questão central em PEGs é o conceito de completude, o qual determina se uma PEG tem sucesso ou falha no processamento de todas as *strings* de entrada. Ford (2004) mostrou que determinar se uma PEG é completa é um problema indecidível. No entanto, ele definiu um critério ao qual assegura que toda PEG que o satisfaça é bem-formada. Ribeiro et al. (2019) argumentam que o critério do Ford pode não ser suficientemente claro e propõe uma abordagem alternativa baseada em tipos para determinar se uma PEG é bem-formada.

Neste trabalho, formaliza-se a abordagem baseada em tipos (RIBEIRO et al., 2019) para verificar terminação de PEGs usando a linguagem PLT Redex (KLEIN et

al., 2012). A formalização neste trabalho é baseada na estratégia de verificação de tipos usando semântica de reescrita (FELLEISEN; FINDLER; FLATT, 2009).

1.1 Objetivos

O objetivo é formalizar, em PLT Redex, uma abordagem para terminação de PEG baseada em tipos usando semântica de reescrita.

1.2 Contribuições

O presente estudo contribui com os seguintes artefatos:

- Uma abordagem para inferência de tipos em PEG usando semântica de reescrita;
- Formalização da abordagem na ferramenta PLT Redex;
- Evidências com testes baseados em propriedades da correção da formalização.

1.3 Estrutura

O restante deste documento está organizado da seguinte forma: o Capítulo 2 abrange a fundamentação teórica utilizada neste trabalho: o que é PEG, sistema de tipos de PEG, semântica de reescrita, Redex, boa-formatura de PEGs e trabalhos relacionados; o Capítulo 3 apresenta a formalização da semântica de reescrita para inferência de tipos de PEG; o Capítulo 4 aborda os testes baseados em propriedades e o Capítulo 5 conclui o trabalho. O código desenvolvido está disponível no endereço ⟨https://github.com/lives-group/redexPEG⟩.

2 Fundamentação Teórica

Este capítulo foi dividido em seções de forma a proporcionar um melhor entendimento sobre as fundamentações do trabalho proposto. A Seção 2.1 define *Parsing Expression Grammar*. A Seção 2.2 discute terminação em PEGs e a Seção 2.3 fecha a fundamentação teórica dos conceitos e definições para a criação do formalismo de uma abordagem baseada em tipo.

A Seção 2.4 apresenta a linguagem PLT Redex, utilizada para formalizar o modelo; e a Seção 2.5 discute os trabalhos relacionados.

2.1 Parsing Expression Grammar

Parsing Expression Grammar (PEG) (FORD, 2004) é um formalismo baseado em reconhecimento para especificação de uma linguagem e validação de um programa. Formalmente, uma gramática PEG G é uma quádrupla (V_N, V_T, R, e_S), em que V_N é um conjunto finito de símbolos não-terminais, V_T é um conjunto finito e disjunto de V_N de símbolos terminais, R é uma função que mapeia não-terminais em *parsing expressions* e e_S é a *parsing expression* inicial. Sejam $a \in V_T$ e $A \in V_N$, o conjunto, P_e, de *parsing expression* é definido recursivamente como:

- $\epsilon \in P_e$;
- $a \in P_e$;
- $A \in P_e$;
- se $e_1 \in P_e$ e $e_2 \in P_e$, então $e_1\ e_2 \in P_e$ e $e_1\ /\ e_2 \in P_e$;
- se $e \in P_e$, então $e* \in P_e$ e $!e \in P_e$.

Ao longo deste documento usamos letras minúsculas para representar símbolos terminais, maiúsculas para não-terminais e e para denotar *parsing expressions*. Como usual, os símbolos podem ser subscritos.

A semântica das *parsing expressions* é dada indutivamente por um julgamento que relaciona uma *parsing expression* e uma entrada com a porção da entrada consumida (Figura 2.1). Usa-se a notação $(e, s) \Rightarrow s_p$ para denotar que a *parsing expression* e consome o prefixo s_p da entrada s. Em caso de falha, utiliza-se a notação $(e, s) \Rightarrow \perp$. A meta-variável r é usada para denotar tanto sucesso (consumo de um prefixo s_p da entrada) como falha ($\perp$).

$$\frac{}{(\epsilon, s) \Rightarrow \epsilon}\ \{Eps\} \qquad \frac{}{(a, as) \Rightarrow a}\ \{ChrS\} \qquad \frac{a \neq b}{(a, bs) \Rightarrow \perp}\ \{ChrF\} \qquad \frac{}{(a, \epsilon) \Rightarrow \perp}\ \{ChrNil\}$$

$$\frac{A \leftarrow e \in R \quad (e, s) \Rightarrow r}{(A, s) \Rightarrow r}\ \{Var\} \qquad \frac{(e_1, s) \Rightarrow \perp}{(e_1\ e_2, s) \Rightarrow \perp}\ \{Cat_{F1}\} \qquad \frac{(e_1, s_p s_r) \Rightarrow s_p \quad (e_2, s_r) \Rightarrow \perp}{(e_1\ e_2, s_p s_r) \Rightarrow \perp}\ \{Cat_{F2}\}$$

$$\frac{(e_1, s_{p1}s_{p2}s_r) \Rightarrow s_{p1} \quad (e_2, s_{p2}s_r) \Rightarrow s_{p2}}{(e_1\ e_2, s_{p1}s_{p2}s_r) \Rightarrow s_{p1}s_{p2}}\ \{Cat_{S1}\} \qquad \frac{(e_1, s_p\ s_r) \Rightarrow s_p}{(e_1\ /\ e_2, s_p\ s_r) \Rightarrow s_p}\ \{Alt_{S1}\} \qquad \frac{(e_1, s_p\ s_r) \Rightarrow \perp \quad (e_2, s_p\ s_r) \Rightarrow r}{(e_1\ /\ e_2, s_p\ s_r) \Rightarrow r}\ \{Alt_{S2}\}$$

$$\frac{(e, s_{p1}s_{p2}s_r) \Rightarrow s_{p1} \quad (e^\star, s_{p2}s_r) \Rightarrow s_{p2}}{(e^\star, s_{p1}s_{p2}s_r) \Rightarrow s_{p1}s_{p2}}\ \{Star_{rec}\} \qquad \frac{(e, s) \Rightarrow \perp}{(e^\star, s) \Rightarrow \epsilon}\ \{Star_{end}\}$$

$$\frac{(e, s_p\ s_r) \Rightarrow s_p}{(!\,e, s_p\ s_r) \Rightarrow \perp}\ \{Not_F\} \qquad \frac{(e, s) \Rightarrow \perp}{(!\,e, s) \Rightarrow \epsilon}\ \{Not_S\}$$

Figura 2.1: Semântica Operacional de PEG.

A regra Eps especifica que a *parsing expression* vazia (ϵ) sempre resulta em sucesso sem consumir a entrada. As regras do terminal especificam que ele pode obter sucesso e consumir a entrada (ChrS), pode falhar (ChrF) e caso a entrada seja vazia, ele também falha (ChrNil). A regra Var indica que o não terminal depende de sua regra; e nas regras da sequência os dois termos devem ser analisados para definir se falha (Cat_{F1} - para caso o primeiro termo falhar - e Cat_{F2} - caso o segundo falhe) ou obtém sucesso (Cat_{S1}). A regra Alt_{S1} resulta em sucesso, caso o primeiro termo consuma algum elemento. A regra Alt_{S2} também obtém sucesso, apenas se o primeiro elemento falhar e o segundo obtiver sucesso. A repetição é um algoritmo guloso que continua consumindo os elementos caso o termo de dentro consuma ($Star_{rec}$) e para de consumir, retornando vazio, caso o elemento de dentro falhe ($Star_{end}$); e as regras do predicado de negação nunca consomem a entrada e negarão o termo de dentro. Se este termo consumir um elemento, então a regra Not_F é utilizada e se este termo falhar, então a regra Not_S será utilizada.

2.2 PEGs Bem-formadas

Ford (2004) provou que determinar se uma PEG sempre finaliza a análise de uma entrada com sucesso ou falha é um problema indecidível. Considerando que a análise sintática é a primeira etapa de uma grande gama de softwares e não deveria entrar em laço infinito, Ford (2004) propõe uma análise conservadora que garante que toda PEG que a satisfaz não entrará em laço infinito durante a análise de qualquer entrada. Uma gramática com tais características é denominada de *bem-formada*. Intuitivamente, uma gramática é bem-formada quando não contém regras diretas ou mutuamente recursivas à esquerda e tem uma repetição gulosa de uma expressão que pode ter sucesso sem consumir símbolos da entrada.

Para definir que uma PEG é bem-formada, Ford (2004) define a relação $\rightharpoonup_G$. Esta relação consiste em pares (e, o) na qual e é uma expressão e $o \in \{0, 1, f\}$. O o indica que a expressão pode obter sucesso sem consumir símbolos de entrada (0), sucesso consumindo uma parte da entrada (1) ou falha (f). Utiliza-se também, a letra s caso um termo possa obter sucesso (0 ou 1). A relação $\rightharpoonup_G$ é definida conforme a seguir:

- *String* vazia ($\epsilon \rightharpoonup 0$): resulta em sucesso com todos os casos e não consome nenhum elemento.

- Terminal ($a \rightharpoonup 1$ ou $a \rightharpoonup f$): o termo pode obter sucesso consumindo um elemento (1) ou pode falhar (f).

- Não terminal ($A \rightharpoonup o$ if $R_G(A) \rightharpoonup o$): é substituído pela sua respectiva regra e só então avaliado.

- Sequência: como este termo possui e_1 e e_2, tem-se que o e_1 será testado primeiro e se obtiver sucesso, testa-se e_2. Se e_2 falhar, toda a sequência falha e se obtiver sucesso, toda a sequência obtém sucesso. Caso e_1 falhe, toda a sequência falha sem testar e_2.

  1. $e_1e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$ and $e_2 \rightharpoonup 0$
  2. $e_1e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$ and $e_2 \rightharpoonup s$
  3. $e_1e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup 1$

4. $e_1e_2 \rightharpoonup f$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup f$

5. $e_1e_2 \rightharpoonup f$ if $e_1 \rightharpoonup f$

- Alternância: este termo também possui e_1 e e_2. O primeiro é testado e, se obtiver êxito, então toda a expressão tem sucesso (1). Se e_1 falhar, então e_2 será testado. Pode obter sucesso e a alternância obterá sucesso ou pode falhar, e dessa forma, a alternância falhará como um todo.

  1. $e_1/e_2 \rightharpoonup s$ if $e_1 \rightharpoonup s$

  2. $e_1/e_2 \rightharpoonup o$ if $e_1 \rightharpoonup f$ and $e_2 \rightharpoonup o$

- Repetição: este termo fará uma repetição gulosa. Poderá obter sucesso e então sairá com sucesso ($e* \rightharpoonup 1$ if $e \rightharpoonup 1$); e caso falhe também sairá com sucesso não consumindo nada ($e* \rightharpoonup 0$ if $e \rightharpoonup f$).

- Predicado *not*: No caso da negação, tem-se a expressão $!e$. O termo interno e é testado e caso obtenha sucesso consumindo ou não a entrada, então o predicado $!e$ falhará ($!e \rightharpoonup f$ if $e \rightharpoonup s$). Se e falhar, então $!e$ obterá sucesso, mas sem consumir nada ($!e \rightharpoonup 0$ if $e \rightharpoonup f$).

O conjunto de *parsing expressions* bem-formada para uma PEG G, WF_G, é definida recursivamente como:

1. $WF_G(\epsilon)$: o termo vazio é bem-formada por si só. Se estiver apenas o ϵ como termo para ser verificado, ele será bem-formado.

2. $WF_G(a)$: o terminal é bem-formado por si só.

3. $WF_G(A)$ if $WF_G(R_G(A))$: o não terminal é bem-formado se a regra associada a ele também for bem-formada. Por exemplo, se um não terminal A tem uma regra associada a ele como sendo um terminal, então ele será bem-formado (A $\rightarrow$ 1, logo o não terminal A é WF_G, pois o terminal, também o é).

4. $WF_G(e_1e_2)$ if $WF_G(e_1)$ and $e_1 \rightharpoonup 0$ implies $WF_G(e_2)$: a sequência se torna bem-formada se o primeiro elemento for bem-formado e caso o primeiro elemento não

consuma nenhuma parte da entrada, então o segundo elemento deverá ser bem-formado.

5. $WF_G(e_1/e_2)$ if $WF_G(e_1)$ and $WF_G(e_2)$: a alternância será bem-formada se ambos os elementos também forem.

6. $WF_G(e*)$ if $WF_G(e)$ and $e \not\Rightarrow 0$: caso o elemento dentro da repetição seja bem-formado e este elemento consuma alguma parte da *string* de entrada, então a repetição, no geral, será bem-formada.

7. $WF_G(!e)$ if $WF_G(e)$: e por último, o predicado de negação só é bem-formado se o elemento dentro dele é bem-formado.

Um gramática PEG G é considerada bem-formada se toda subexpressão de G for bem-formada.

2.3 Sistema de Tipos de PEGs

Ford (2004) propõe a relação WF_G como solução à recursão infinita, isto é, PEGs que não contém regras diretas ou indiretas com recursão à esquerda e nem expressões exitosas que não consomem nenhum símbolo e estão dentro de uma repetição. O trabalho apresentado neste estudo é uma alternativa ao conceito de boa formatura de Ford. O que Ford propõe para resolver o *loop* infinito, o trabalho resolve com sistema de tipos.

A formação de um tipo para PEG se dá por um par composto de um *booleano* e uma lista de não terminais. O *booleano*, também chamado de anulável ou $\tau.null$, indica se o termo pode obter sucesso consumindo alguma parte da entrada (*true* para: obtém sucesso sem consumir e *false* para: obtém sucesso consumindo). Já a lista de não terminais, também chamada de *head-set*, é o conjunto de símbolos que aparece imediatamente à esquerda de uma regra, conforme a conjunção abaixo:

A Figura 2.3 mostra como a lista de não terminais de um tipo de uma PEG é contruída. Para os termos *string* vazia e terminal, o *head-set* será nulo. Para o não terminal, seu *head-set* é definido como ele próprio e o *head-set* da regra associada a ele. Para a lista da repetição e do predicado de negação basta verificar a lista da expressão

$$
\begin{array}{lcl}
head(\epsilon) & = & \emptyset \\
head(a) & = & \emptyset \\
head(A) & = & \{A\} \cup head(R(A)) \\
head(e_1\, e_2) & = & \begin{cases} head(e_1) \cup head(e_2) & \text{if } e_1 \rightharpoonup 0 \\ head(e_1) & \text{otherwise} \end{cases} \\
head(e_1 \,/\, e_2) & = & head(e_1) \cup head(e_2) \\
head(e^\star) & = & head(e) \\
head(!\, e) & = & head(e)
\end{array}
$$

Figura 2.2: Definição de *head-set* para PEG

de dentro. Para a alternância é o mesmo princípio, apenas com a união dos *head-sets* dos dois termos e para sequência deve-se verificar se o primeiro termo é anulável ou não. Caso seja, então o *head-set* será a união dos *head-sets* dos dois termos e caso não seja anulável, apenas o *head-set* do primeiro termo é necessário.

Seguindo Ribeiro et al. (2019), τ denota um tipo arbitrário, $\tau.null$ denota o campo *booleano* de τ, e $\tau.head$ o conjunto de variáveis que podem aparecer como o primeiro símbolo da PEG de τ. A notação $\langle b, S\rangle$ denota um tipo τ formado por um *booleano* b e um conjunto S. No sistema de tipos, utiliza-se as seguintes operações:

$$
\begin{array}{lcl}
b \Rightarrow S & = & \text{if } b \text{ then } S \text{ else } \emptyset \\
\tau_1 \otimes \tau_2 & = & \langle \tau_1.null \wedge \tau_2.null, \tau_1.head \cup (\tau_1.null \Rightarrow \tau_2.head)\rangle \\
\tau_1 \oplus \tau_2 & = & \langle \tau_1.null \vee \tau_2.null, \tau_1.head \cup \tau_2.head\rangle \\
!\,\tau & = & \langle true, \tau.head\rangle \\
\langle false, S\rangle^\star & = & \langle true, S\rangle
\end{array}
$$

Figura 2.3: Funções auxiliares para formação de tipos

A primeira operação é um teste *booleano* que retorna o conjunto S sempre que a condição for verdadeira. A operação de produto ($\tau_1 \otimes \tau_2$) combina dois tipos e faz a conjunção de seus campos *booleanos* e a união de seus respectivos *headsets*, caso $\tau_1.null$ seja verdadeiro. A operação de coproduto ($\tau_1 \oplus \tau_2$) é semelhante ao produto, mas ao invés da conjunção dos campos *booleanos*, é feita a disjunção. A operação *not* apenas altera o campo *booleano* do tipo para verdadeiro e mantém seu *head-set*. A última operação é a *star* Kleene para repetição, que também define o campo *booleano* do tipo como verdadeiro. Porém, o Kleene só é definido para tipos τ tais que $\tau.null = false$, sendo indefinido quando $\tau.null = true$.

Cardoso et al. (2022) além de descrever a relação acima, também descreve regras para a composição de um sistema de tipos para todos os termos. Observe que nessa

relação tem-se a inclusão de um contexto de tipo Γ que contém pares de não terminais e seus respectivos tipos (A, τ).

$$\frac{}{\Gamma \vdash \epsilon : \langle true, \emptyset \rangle} \quad \frac{}{\Gamma \vdash a : \langle false, \emptyset \rangle} \quad \frac{\Gamma(A) = \tau \quad A \notin \tau.head}{\Gamma \vdash A : \langle \tau.null, \tau.head \cup A \rangle}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \,/\, e_2 : \tau_1 \oplus \tau_2} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash !\, e : !\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma \vdash e : \tau \quad \tau.null = false}{\Gamma \vdash e^\star : \tau^\star}$$

Figura 2.4: Definição do sistema de tipos

A primeira regra do sistema de tipos especifica que a PEG ϵ tem o tipo $\langle true, \emptyset \rangle$. A regra de um terminal diz que seu tipo é $\langle false, \emptyset \rangle$, já que ela não obtém sucesso sem consumir um símbolo e nenhuma variável pode aparecer como a "cabeça" de sua árvore de reconhecimento, ou seja, seu *head-set* é $\emptyset$. A expressão de alternância (e_1/e_2) tem tipo $\tau_1 \oplus \tau_2$, onde $\Gamma \vdash e_1 : \tau_1$ e $\Gamma \vdash e_2 : \tau_2$.

O tipo de uma expressão de sequência é dado pela operação de produto em seus tipos de subexpressões, e a operação estrela $(\star)$ só é bem-formada se sua expressão subjacente não for bem-sucedida sem consumir um símbolo. O predicado *not* sempre terá o tipo $\langle true, \tau.head \rangle$ sempre que sua subexpressão for bem-formada.

Finalmente, um não terminal é bem-tipado somente se seu tipo estiver no contexto e não pertencer ao seu *head-set*.

Exemplificando o que foi descrito, considera-se a gramática como sendo formada por duas variáveis R e S. R tem como regra associada S e S tem como regra associada, uma repetição do terminal 1: $(R\ S)\ (S(1\ *))$. Para a formação do tipo relativo a R, deve-se considerar a regra do *head-set* relativo ao não terminal. O *head-set* de R ficará da seguinte forma:

$$
\begin{aligned}
head(a) &= \emptyset \\
head(A) &= \{A\} \cup head(R(A)) \\
head(e^{\star}) &= head(e)
\end{aligned}
$$

$$
\begin{aligned}
head(R) &= \{R\} \cup head(S) \\
head(R) &= \{R\} \cup \{S\} \cup head(1*) \\
head(R) &= \{R\} \cup \{S\} \cup head(1) \\
head(R) &= \{R\} \cup \{S\} \cup \emptyset \\
head(R) &= \{R\} \cup \{S\} \\
head(R) &= \{R,\ S\}
\end{aligned}
$$

Figura 2.5: Exemplo relativo ao tipo de R

O campo anulável a esse não terminal corresponde ao campo anulável do tipo da regra associda a ele, que por sua vez, é o tipo da regra associada a S. A repetição $(1\ *)$ tem como tipo $\langle false,\ \emptyset\rangle$, então o campo anulável de R é $false$ também. O tipo concreto ao final da avaliação das regras ficará $\langle false,\ (R,\ S)\rangle$

Com a definição do sistema de tipos, é possível mapear todos os tipos para todos os termos de uma PEG.

2.4 Racket e Redex

Racket é um dialeto da família Lisp de linguagens de programação. Lisp é uma das linguagens de programação mais antiga, desenvolvida em 1958 no Instituto de Tecnologia do Massachusetts. Foi projetado, inicialmente, como um sistema de notação matemática para computadores, mas evoluiu para uma linguagem de programação completa que foi usada por pioneiros da Ciência da Computação.

Muitos dos conceitos fundamentais da computação foram desenvolvidos pela primeira vez em Lisp. Inclui condicionais, digitação dinâmica, recursão, coleta de lixo e compiladores auto-hospedados.

Já Racket tem seu início em 1995, quando uma empresa chamada PLT começou a desenvolver materiais de aprendizagem para programação. O objetivo era construir

um ambiente que pudesse ser utilizado para ensinar programação Scheme para iniciantes, e portanto, criaram uma ramificação chamada PLT Scheme. Felleisen, Findler e Flatt (2009) publicaram um livro sobre PLT Scheme que traz todas as problemáticas, conceitos e explicações sobre o que é a linguagem e como utilizá-la.

Ao longo dos anos, o PLT Scheme adicionou mais e mais recursos. Em 2010, a versão 5.0 foi lançada e o projeto foi renomeado para Racket. Como Racket é uma linguagem que permite o aprimoramento de seus recursos e a adição de recursos criados, ela foi escolhida para tornar este trabalho aplicável e funcional.

Segundo, Felleisen, Findler e Flatt (2009), Redex, como um modelo básico, consiste em uma gramática de árvore regular e um conjunto de regras de redução. Utiliza-se uma linguagem de domínio específico para formular a sintaxe e semântica da linguagem que se quer especificar. Primeiro, define-se a linguagem, especificando a sintaxe; é preciso escolher um nome para linguagem e adicionar seus termos e regras que formarão posteriormente a semântica.

A biblioteca de Redex dispõe de uma série de ferramentas para exploração da linguagem, como teste, *debugging*, checagem de relações de redução, tipagem e outros tipos de ferramentas que são capazes de formular uma linguagem completa, apenas com uma biblioteca.

2.5 Trabalhos Relacionados

O primeiro trabalho relacionado é descrito no artigo de Krishnaswami e Yallop (2019), que expõe uma abordagem algébrica tipada para análise de desempenho de tempo, de transferência de dados e análise geral dos *parsers* combinatórios juntamente com a construção de gramática e sistema de tipos. Os resultados foram satisfatórios para algumas linguagens em específico, mas encontrou problemas na leitura e conversão das *strings* de entrada.

O trabalho de Medeiros, Mascarenhas e Ierusalimschy (2014) trata de um problema recorrente em analisadores, no geral: recursão à esquerda, gerando uma recursão infinita. A causa principal de recursão à esquerda em PEG é os não terminais. Sendo assim, é reafirmado que todos os não terminais em PEG devem ter apenas uma regra.

Portanto, Medeiros, Mascarenhas e Ierusalimschy (2014) elaboram um pensamento baseado na limitação da recursão à esquerda. A extensão da teoria promovida por eles é baseada na recursão à esquerda limitada, onde o número de usos recursivos à esquerda que um não-terminal pode ter é restringido garantindo o término; e usa-se um processo iterativo para encontrar o menor limite que fornece a correspondência mais longa para um uso específico do não terminal.

Ribeiro et al. (2019) se relaciona diretamente com o presente estudo quando descreve a formalização de um sistema de tipos para analisar gramáticas de expressão (PEG) que é equivalente à definição original baseada em pontos fixos de terminação de PEGs de Ford (2004). É usada uma definição de sistema de tipos, que foi descrita nas seções anteriores para implementar uma semântica funcional de passo grande para PEGs digitadas usando a linguagem de programação Agda.

Neste trabalho, é criado um sistema de tipos que pode ser utilizado como alternativa à boa formação de PEGs (PEGs bem-formadas). É mostrado que o sistema de tipos é equivalente ao predicado de boa formação de Ford (2004). Um interpretador deste sistema de tipos para PEGs em Agda é desenvolvido.

E por fim, se relacionando com a seção de testes, tem-se que o teste baseado em propriedade (PBT) é uma técnica para validar o código em relação a uma especificação executável, gerando dados de teste automaticamente. Blanco, Miller e Momigliano (2019) apresentam uma reconstrução de prova teórica desse estilo de teste para especificações relacionais e emprega a estrutura do *Foundational Proof Certificate* para descrever geradores de teste.

Este artigo, então, descreve certos tipos de "esquemas de prova" que podem ser usados para descrever várias estratégias de geração comuns de teste na literatura PBT, variando de aleatório a exaustivo, incluindo sua combinação.

3 Formalização da Abordagem

A formalização da inferência de tipo se dá a partir de duas fases distintas e interligadas: geração de restrições e redução da restrição. Como toda a abordagem formalizada em Redex se baseia na criação de uma linguagem e na resolução dela com semântica de passo pequeno ou semântica de passo grande, a inferência de tipo não será diferente. Na primeira Seção (3.1) deste capítulo aborda-se a definição da linguagem de restrições e operações sobre tipos. A Seção 3.2 explica como se dá a formação das restrições a partir de um termo PEG. A Seção 3.3 explica as regras associadas às restrições para serem reduzidas logo após na Seção 3.4.

3.1 Definição da Linguagem

O primeiro passo para iniciar a formalização é definir uma linguagem de restrição. A linguagem de restrições, abaixo listada, descreve tipos básicos (τ) na linha 2 da listagem abaixo, termos (t) na linha 3 e fórmulas (C) na linha 4. O tipo τ pode ser uma variável de tipo α; ou um tipo concreto ($b\ S$). Foi necessário complementar a linguagem de restrições com as operações sobre tipos descritas na Figura 2.3 e uma nova operação de clone. A operação ($clone\ \tau\ x$) foi inserida para capturar as restrições referentes ao tipo de variável e diminuir a quantidade de reduções.

Os termos t podem ser variáveis (não-terminais) ou tipos e por fim, as fórmulas C, que podem ser a equivalência entre dois termos ($t\ \equiv\ t$), a conjunção de duas fórmulas ($\wedge\ C\ C$), ou uma associação de um tipo a uma variável, ($def\ x\ :\ \tau\ in\ C$).

Os contextos ψ (linha 10) e ϕ (linha 11) contém, respectivamente, o ambiente de tipos de variáveis, isto é, o mapeamento de não terminais para seus respectivos tipos e a substituição, mapeamento de variáveis de tipos para tipos concretos.

A cada passo, uma equação é selecionada, resolvida por unificação e o resultado é composto com a substituição corrente. O processo é iterado até que todas as restrições tenham sido resolvidas ou até que não seja mais possível resolver nenhuma restrição.

Abaixo é mostrado a definição da linguagem de restrições com PLT Redex, ilustrando o que foi mencionado acima:

```
1  (define-extended-language Typed-Peg Peg
2    [τ ::= α (b S) (× τ τ) (+ τ τ) (⋆ τ) (! τ) (clone τ x)]
3    [t ::= x τ]
4    [C ::= b (t ≡ t) (∧ C C) (def x : τ in C)]
5    [CEval ::= hole (∧ CEval C) (∧ C CEval)]
6    [α ::= (∨ natural)]
7    [S ::= (x ...)]
8    [b ::= #t #f]
9    [x ::= variable-not-otherwise-mentioned]
10   [ψ ::= ((x τ)...)]
11   [ϕ ::= ((α τ)...)])
```

Listing 3.1: Linguagem de Restrições

3.2 Formação das Restrições

Inicialmente a PEG deve ser transformada em restrições. Este processo é feito por meio de uma metafunção em Redex.

Durante o processo de conversão de um termo PEG em uma fórmula de restrição C será necessário criar variáveis de tipo novas, que não tenham aparecido livres na fórmula. Para tanto utiliza-se um contador para gerar o nome dessas variáveis.

Seguindo a formalização do artigo de Cardoso et al. (2023), a cada termo partir de PEG atribui-se um tipo τ e o resultado é composto por um par: fórmula C e um novo valor para o contador de variáveis.

A metafunção que irá converter o termo PEG em uma fórmula C é chamada de $tcMonad$ e utiliza como parâmetros o termo e, um tipo τ e um número natural inicial 0: $(tcMonad : e\ \tau\ natural \rightarrow (C\ natural))$.

Dentro desta metafunção, a PEG será avaliada de acordo com cada caso do termo e. No caso em que e é o termo ϵ, gera-se uma equivalência entre o tipo dado e um tipo concreto que será formado por um *booleano* #t e um *headset* vazio (o tipo de ϵ). Neste termo, o natural não é alterado, apenas é passado como resultado, pois nenhuma variável

de tipo foi acrescentada à restrição.

$$\epsilon\ ,\ (\tau\ natural)\ \rightarrow\ ((\tau\ \equiv\ (\#t\ ()))\ natural)$$

A restrição relativa a um natural é descrita a seguir como a equivalência do tipo passado como parâmetro com o tipo $(\#f\ ())$, que é o tipo do terminal da PEG. O número natural também não é acrescentado pois não há criação de nova variável.

$$natural_1\ ,\ (\tau\ natural)\ \rightarrow\ ((\tau\ \equiv\ (\#f\ ()))\ natural)$$

Para gerar restrições para a alternância, define-se duas novas variáveis de tipo, α_1 e α_2, passadas como argumentos para a geração de restrições para e_1 e e_2, respectivamente. A operação de coproduto é utilizada como tipo resultante da restrição $(+\ \alpha_1\ \alpha_2)$. E a conjunção das restrições de e_1 (C_1) e a conjunção de e_2 (C_2) forma a restrição geral da alternância.

$$(e_1\ /\ e_2)\ ,\ (\tau\ natural)\ \rightarrow\ ((\wedge\ (\wedge\ C_1\ C_2)\ (\tau\ \equiv\ (+\ \alpha_1\ \alpha_2)))\ natural)$$

A geração de restrição da sequência segue o mesmo padrão da restrição de alternância. A única diferença é a operação sobre os tipos - que será de produto.

$$(e_1\ e_2)\ ,\ (\tau\ natural)\ \rightarrow\ ((\wedge\ (\wedge\ C_1\ C_2)\ (\tau\ \equiv\ (\times\ \alpha_1\ \alpha_2)))\ natural)$$

Outra regra importante que deve ser descrita é a regra da repetição (e_1*). Ela é representada como uma conjunção entre o resultado das restrições do conjunto interno da repetição (e_1) e o tipo de uma repetição representado no programa pela $(\star\ \tau)$, já mencionado anteriormente. Portanto, no termo $(*\ e_1)$, é feita as restrições de e_1 juntamente com o tipo de e_1 dentro da relação $\star$. O termo de negação segue o mesmo fluxo.

$$*\ e_1, (\tau\ natural) \rightarrow (\wedge\ C\ (\tau\ \equiv\ (\star\ v\ natural)))$$

$$! \; e_1, (\tau \; natural) \rightarrow (\wedge \; C \; (\tau \; \equiv \; (! \; v \; natural)))$$

A conversão de uma gramática PEG para uma fórmula C é realizada pela metafunção "gc1Monad", criada na formalização em Redex. Os parâmetros dessa metafunção são: a gramática G, uma fórmula C, o ambiente de variáveis ψ e um contador de variáveis chamado de *natural*. E o resultado é composto pelo novo ambiente ψ com o mapeamento relativo aos não terminais, uma nova fórmula C correspondente à restrição e um novo valor para o contador de variáveis de tipo.

$$\psi \; , \; G \; , \; C \; , \; natural \rightarrow \; ((\psi \; C) \; natural)$$

Como a gramática em PEG descrita em Redex é uma lista de pares de variável e termo $(G \; ((x \; e) \; ...))$, verifica se a gramática é vazia e se isto for verdade retornam-se os proŕios C, ψ e *natural*, passados como parâmetro.

$$\psi \; , \; () \; , \; C \; , \; natural \; \rightarrow \; ((\psi \; C) \; natural)$$

Caso contrário, isto é, a gramática é composta por um ou mais pares de não terminal e termo: insere-se o não terminal no contexto ψ com a metafunção $\psi cons$ e emprega-se a metafunção "tcMonad", também criada na formalização, no termo relativo ao não terminal, obtendo-se uma nova fórmula C_1. Faz-se a conjunção de C_1 com C para acrescentar este novo resultado ao conjunto de restrições geral. Finalmente, a função "gc1Monad" é chamada recursivamente para continuar percorrendo as demais regras da gramática da PEG.

Utiliza-se um exemplo com a gramática composta por R: ($(R \; 5)$), para elucidar o leitor. Os parâmetros da função são uma lista vazia que compõe o ambiente ψ, a gramática, uma fórmula C qualquer $(\#t)$ e um número natural (0).

$$(gc1Monad \; (\;) \; ((R \; 5)) \; \#t \; 0)$$

A gramática é inserida na metafunção "gc1Monad" e como ela possui um elemento

(R) cairá na segunda regra descrita acima $(\psi\ ,\ ()\ ,\ C\ ,\ natural\ \rightarrow\ ((\psi\ C)\ natural))$. R será colocado no ψ juntamente com sua variável de tipo (fresh) criada automaticamente com o número natural, também passado como parâmetro. Logo, o ambiente ψ ficará $(\ (R\ (v\ 0))\ )$. O termo 5 é inserido no "tcMonad" para formular sua respectiva restrição e o número natural é incrementado 1, pois uma nova variável foi criada $(v\ 0)$. O restante da gramática é passado como parâmetro para o "gc1Monad" novamente, junto com o novo resultado C, ψ e 1 (número natural). Como a gramática desta nova chamada de função será vazia, o resultado é retornado.

$$'((((R\ (v\ 0)))\ ((v\ 0)\ \equiv\ (\#f\ ()))))\ 1)$$

Com as duas metafunções descritas acima é possível formar a terceira que une as restrições da gramática e as restrições da expressão da PEG para utilizar esta união dentro da semântica de passo pequeno (reduction-relation). Além disso, ambos os contextos (ψ e ϕ) precisam estar junto com a restrição. Para tanto, formula-se uma terceira função que faz essa avaliação das expressões (gramática e termo) e as coloca junto do contexto de variável e o contexto de tipo. O contexto de tipo é atribuído a uma lista vazia inicialmente e o contexto de variável é calculado pela metafunção "gc1Monad" vista anteriormente. A gramática é colocada em "gc1Monad" e o termo em "tcMonad" e une-se os resultados para formar os parâmetros corretos para a relação de redução.

A semântica de passo pequeno foi escolhida nesta formalização para apresentar de forma prática os resultados obtidos com o *parsing* da expressão de restrições obtida na especificação anterior. Então, conforme a Figura abaixo, desenvolvida pelo artigo Cardoso et al. (2023) é possível interpretar a expressão de restrição e verificar se aquela expressão faz parte da gramática ou não.

$$
\begin{array}{lll}
(1) & \langle \psi, \phi, \textbf{def } A : \tau \textbf{ in } C \rangle & \rightarrow \langle \psi[A \mapsto \phi\tau], \phi, C \rangle \\
(2) & \langle \psi, \phi, (\exists \overline{\alpha}.C_1) \wedge C_2 \rangle & \rightarrow \langle \psi, \phi, \exists \overline{\alpha}.C_1 \wedge C_2 \rangle, \\
 & & \textbf{if } \overline{\alpha} \# \mathrm{fv}(C_2) \\
(3) & \langle \psi, \phi, \tau_1 \equiv \tau_2^\star \rangle & \rightarrow \langle \psi, \phi, \textbf{false} \rangle, \\
 & & \textbf{if } \tau_2.\mathrm{null} = \mathrm{true} \\
(4) & \langle \psi, \phi, A \equiv \tau \rangle & \rightarrow \langle \psi, \phi, \textbf{false} \rangle, \\
 & & \textbf{if } A \in (\phi(\psi A)).\mathrm{head} \\
(5) & \langle \psi, \phi, C \wedge \textbf{false} \rangle & \rightarrow \langle \psi, \phi, \textbf{false} \rangle \\
(6) & \langle \psi, \phi, \textbf{false} \wedge C \rangle & \rightarrow \langle \psi, \phi, \textbf{false} \rangle \\
(7) & \langle \psi, \phi, C \wedge \textbf{true} \rangle & \rightarrow \langle \psi, \phi, C \rangle \\
(8) & \langle \psi, \phi, \textbf{true} \wedge C \rangle & \rightarrow \langle \psi, \phi, C \rangle \\
(9) & \langle \psi, \phi, \alpha \equiv \tau \rangle & \rightarrow \langle \psi, \phi[\alpha \mapsto \tau], \textbf{true} \rangle \\
(10) & \langle \psi, \phi, \tau \equiv \alpha \rangle & \rightarrow \langle \psi, \phi[\alpha \mapsto \tau], \textbf{true} \rangle \\
(11) & \langle \psi, \phi, \langle b_1, S_1 \rangle \equiv \langle b_2, S_2 \rangle \rangle & \rightarrow \langle \psi, \phi, b_1 \equiv b_2 \wedge S_1 \equiv S_2 \rangle
\end{array}
$$

Figura 3.1: Resolução de restrição

Observa-se um exemplo abaixo para a PEG (/ R 5), onde a gramática é (R (* 0)). Deve-se formar uma semântica de inferência de tipo na seguinte ordem: ambiente ψ, ambiente ϕ e as restrições C. Primeiro, a inferência do termo (/ R 5) é feita. De acordo com as regras da função de restrição de expressão (*tcMonad*), a regra de alternância corresponde a conjunção das subexpressões. O tipo relativo a essa expressão seria os tipos desses termos sozinhos aplicados a regra de tipo +.

```
1. (tcMonad (/ R 5) (v 0) 1) → (∧ (∧ (tcMonad R (v 1) 2) (tcMonad 5)))

2. (τ ≡ (+ τ(R) τ(5)))

3. (tcMonad R (v 1) 2) → (∧ (R ≡ (v 2)) ((v 1) ≡ (clone (v 2) R))) 3

4. (tcMonad 5 (v 3) 4) → ((v 3) ≡ (#f ())) 4
```

Listing 3.2: Restrição do termo

Portanto o *tcMonad* para (/ R 5) é:

```
((∧
    (∧
        (∧
            (R ≡ (v 2))
            ((v 1) ≡ (clone (v 2) R)))
        ((v 3) ≡ (#f ())))
    ((v 0) ≡ (+ (v 1) (v 3))))
4)
```

Listing 3.3: Resultado da restrição do termo

Após a inferência do termo, faz-se a inferência da gramática. O não-terminal R é inserido no ambiente ψ e a interpretação de seu respectivo termo: (* 0) é feita, utilizando a variável de partida natural 4 (anterior).

```
1. (gc1Monad () ((R (* 0))) #t 4) →
                    (gc1Monad ((R (v 4))) () (tcMonad (* 0) (v 4) 5) ?)

2. (tcMonad (* 0) (v 4) 5) → ((∧ ((v 5) ≡ (#f ())) ((v 4) ≡ (⋆ (v 5)))) 6)

→ ((((R (v 4))) (∧ ((v 5) ≡ (#f ())) ((v 4) ≡ (⋆ (v 5))))) 6)
```

Listing 3.4: Restrição da gramática

No conjunto, a restrição ficará da seguinte forma:

```
(((R (v 4))) ()
  (∧
    (∧ ((v 5) ≡ (#f ())) ((v 4) ≡ (⋆ (v 5))))
      (∧
        (∧
          (∧ (R ≡ (v 2)) ((v 1) ≡ (clone (v 2) R)))
            ((v 3) ≡ (#f ())))
        ((v 0) ≡ (+ (v 1) (v 3)))))
  )
)
```

Listing 3.5: Resultado da retrição da gramática

Partindo da formação da restrição acima, definimos a relação de redução.

3.3 Resolução das Restrições

O contexto de redução $CEval$ determina quais sub-expressões da linguagem de restrições podem ser reduzidas.

$$[CEval \ ::= \ hole \ (\wedge \ CEval \ C) \ (\wedge \ C \ CEval)]$$

A definição desse contexto deixa claro que a aplicação das regras de redução pode ocorrer de qualquer lado do operador de conjunção, mas não em uma definição de tipo de uma variável.

A Figura 3.1 acima fornece 11 regras para resolução das restrições que foram criadas na seção anterior. Uma vez que a geração de restrições foi feita, o algoritmo deve resolver essas restrições para obter um resultado coerente com a linguagem: uma fórmula C informando se o termo PEG analisado faz parte ou não da linguagem e o tipo relativo aos não terminais ψ.

Abaixo, segue uma listagem de todas as regras e suas explicações:

1. A primeira regra diz que uma construção def de não terminal A é resolvida inserindo-se A no ambiente de não terminais ψ e resolve-se C.

2. A segunda regra apenas move uma variável ligada a um termo correspondente a existência $\exists$ para o lado mais a esquerda, desde que tal variável não ocorra livre em C_2.

3. A terceira regra é a equivalência entre um tipo τ_1 e outro tipo τ_2 em uma repetição. Essa regra só será falsa ($C \ \rightarrow \ \#f$) se o *booleano* de τ_2 for verdadeiro, pois significa que o termo entraria em recursão infinita dentro da linguagem.

4. A regra quatro faz a equivalência entre um não terminal A e um tipo τ. Seu resultado também será falso caso o não terminal A esteja em seu próprio *head-set*, pois também significa que o termo relativo ao não terminal faz recursão à esquerda e consequentemente, entre em *loop* infinito.

5. As conjunções no geral obedecem à regra lógica do E. Se algum dos termos for

falso, o resultado será falso, caso algum dos termos for verdadeiro, o resultado será a redução do outro termo (regras 5, 6, 7 e 8).

6. As regras 9 e 10 também obedecem a um padrão, pois é a equivalência de uma variável de tipo α para um tipo τ (9) e vice-versa, equivalência entre um tipo τ para uma variável de tipo α (10). Como se tem o ambiente de tipos ϕ, então, a variável de tipo é inserida em ϕ e é mapeada para o tipo ao qual equivale (nos dois casos).

7. A última regra traz tipos concretos (com o *booleano* e o *head-set*) e faz a equivalência de dois tipos concretos diferentes. Essa restrição será reduzida para uma conjunção entre a equivalência dos *booleanos* com a equivalência dos *head-sets*.

3.4 Definição da Relação de Redução

A relação de redução em Redex, é definida por um conjunto de regras de redução na forma $(-->\ (\psi\ \phi\ C_1)\ (\psi\ \phi\ C_2))$, onde $(\psi\ \phi\ C_1)$ é o de termo a ser reescrito, formado pelo contexto de tipos ψ, a substituição ϕ e uma fórmula C_1, e o termo $(\psi\ \phi\ C_2)$ resultante da reescrita. Abaixo exemplifica-se uma das regras referente a tipagem da repetição, que julga que a repetição de uma expressão que pode aceitar ϵ é mal-tipada.

```
(--> (ψ φ (in-hole CEval ( t ≡ (⋆ (#t S)))))
     (ψ φ #f)
     "r-3")
```

Listing 3.6: Cláusula de Redução

A regra tipa uma operação $\star$ cujo campo *nullable* não poderá ser falso, caso contrário, a PEG entrará em recursão infinita (regra 3 da Figura 3.1). Como o objetivo é evitar essa situação, quando o tipo resultante da operação $\star$ de uma PEG anulável é falso, independentemente da lista de não-terminais, o resultado da redução também será falso, demonstrando que aquela PEG não pertence à linguagem, pois entrará em *loop* infinito.

A primeira regra é genérica e feita para que o termo C seja simplificado ao máximo antes de ser realmente reduzido conforme as regras da Figura 3.1.

Esta regra utiliza a metafunção "Csimplify" que engloba as regras 1, 2, 9 e 10 da Figura 3.1 além de simplificar com as operações de tipo descritas acima ($+\ \tau\ \tau$, $\times\ \tau\ \tau$,

etc).

A regra do não terminal (4) é dividida em duas regras na relação de redução. Primeiro, é verificado se o tipo associado ao não terminal é composto por um *booleano* e um *head-set* (b S). Caso não esteja neste formato, e o tipo seja uma variável de tipo (v 1, v 0, v 4, etc), substitui-se a variável de tipo pelo seu respectivo tipo que se encontra no ambiente ϕ. Caso esteja no formato *booleano* e *head-set*, o resultado é falso se A pertence ao próprio *head-set*.

O restante das regras é exatamente como demonstradas na Figura 3.1.

4 Resultados e Testes

A avaliação da semântica foi realizada usando um gerador de PEGs bem-formadas Cardoso et al. (2022). Com a biblioteca de testes *rackcheck*, baseada em propriedade, verifica se os tipos inferidos pela nossa implementação é o mesmo obtido pela ferramenta usando o Z3 de (CARDOSO et al., 2023). A seguir, encontra-se o código em Racket para testar esta propriedade.

```
(define-property type-checks([peg  (gen:peg 3 5 2)])
  (compare-types (get-type-of-typed-peg peg)
                 (get-type-of-genConstraint peg)))
```

Listing 4.1: Função de Teste

Exemplificando o que foi dito acima, seja o termo $((R\ (*\ 0)\ \oslash)\ \ (/\ R\ 5))$, em que a gramática contém a regra R associada a uma expressão de repetição $(*\ 0)$; e a expressão a ser analisada é a alternativa $(/\ R\ 5)$. A PEG é inserida na função para geração de restrições que gera a seguinte restrição:

```
(((R (v 4)))                                                    ψ
  ()                                                            φ
  (∧
   (∧ ((v 5) ≡ (#f ())) ((v 4) ≡ (⋆ (v 5))))
   (∧ (∧ (∧ (R ≡ (v 2)) ((v 1) ≡ (clone (v 2) R)))
            ((v 3) ≡ (#f ())))
      ((v 0) ≡ (+ (v 1) (v 3))))))                              C
```

Listing 4.2: Restrição

A primeira lista se refere ao ambiente ψ que possui a lista de não-terminais da gramática associada ao seu respectivo tipo. A segunda lista se refere a ϕ que inicialmente é vazia. E a última lista é composta pelas restrições relativas à gramática e ao termo. O tipo $(v\ 0)$ indica o tipo relativo a 5, $(v\ 5)$ e $(v\ 4)$ se associam ao tipo de R. E a conjunção do tipo de 5 junto da equivalência de R formam a restrição para a alternância.

Essa restrição é inserida na relação de redução composta pelas regras da lingua-

gem Typed-Peg mostrada anteriormente, cujo resultado obtido é composto pelos ambientes ψ, ϕ e uma fórmula C que indica se o termo faz parte ou não da linguagem:

```
'((((R (#t ()))))                                                    ψ
   (((v 0) (#t (R))) ((v 1) (#t (R))) ((v 2) (#t ())))               φ
   ((v 3) (#f ())) ((v 4) (#t ())) ((v 5) (#f ())))
   #t))                                                              C
```

Listing 4.3: Redução

No teste de propriedade acima, obtém-se o resultado da mesma PEG para a biblioteca "typed-peg" (utilizando o Z3) e o tipo relativo aos não-terminais presentes na gramática é extraído. Este resultado é comparado com o ψ obtido acima e conclui-se que eles são iguais, mostrando que o resultado de duas bibliotecas diferentes em relação a uma mesma PEG obtém a mesma saída.

5 Conclusão

As PEGs, introduzidas por Ford em 2004, oferecem uma formalização alternativa à gramáticas livres de contexto para descrever linguagens. Um dos desafios de PEGs é determinar se não entrará em laço infinito para qualquer entrada. Ford (2004) demonstrou que este problema é indecidível e definiu um critério satisfatório que se uma gramática a satisfaz, seguramente não entrará em laço infinito. No entanto, a abordagem do Ford basea-se em algoritmos de ponto-fixo de duas etapas, o qual Ribeiro et al. (2019) argumentam que não é claro. Como alternativa, eles propõem uma abordagem baseada em tipos. Neste trabalho, foi apresentada uma formalização do algoritmo de inferência de tipos proposto por (CARDOSO et al., 2023) usando uma abordagem baseada na semântica de reescrita (KUAN; MACQUEEN; FINDLER, 2007).

Usando uma semântica executável implementada em PLT Redex (FELLEISEN; FINDLER; FLATT, 2009), executa-se testes que fornecem evidências que a formalização está correta. Planeja-se estender o conjunto de testes, usando as técnicas de testes baseados em propriedades, para aumentar a confiança na correção da formalização, analisando cobertura de código - não desenvolvida neste projeto.

Bibliografia

BLANCO, R.; MILLER, D.; MOMIGLIANO, A. Property-based testing via proof reconstruction. In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. ACM, 2019. Disponível em: ⟨https://doi.org/10.1145\%2F3354166.3354170⟩.

CARDOSO, E.; PEREIRA, D.; PAULA, R. D.; REIS, L.; RIBEIRO, R. A type-directed algorithm to generate random well-formed parsing expression grammars. In: *Anais do XXVI Simpósio Brasileiro de Linguagens de Programação*. Porto Alegre, RS, Brasil: SBC, 2022. p. 8–14. ISSN 0000-0000. Disponível em: ⟨https://sol.sbc.org.br/index.php/sblp/article/view/22018⟩.

CARDOSO, E. M.; PAULA, R. D.; PEREIRA, D.; REIS, L.; RIBEIRO, R. G. Type-based termination analysis for parsing expression grammars. In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2023. Disponível em: ⟨https://doi.org/10.1145\%2F3555776.3577620⟩.

CARDOSO, E. M.; PEREIRA, D. F.; PAULA, R. S. M. A. D.; REIS, L. V. D. S.; RIBEIRO, R. G. A type-directed algorithm to generate random well-formed parsing expression grammars. In: *Proceedings of the XXVI Brazilian Symposium on Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2022. (SBLP '22), p. 8–14. ISBN 9781450397445. Disponível em: ⟨https://doi.org/10.1145/3561320.3561326⟩.

FELLEISEN, M.; FINDLER, R. B.; FLATT, M. *Semantics engineering with PLT Redex*. [S.l.]: Mit Press, 2009.

FORD, B. Parsing expression grammars. *ACM SIGPLAN Notices*, Association for Computing Machinery (ACM), v. 39, n. 1, p. 111–122, jan 2004. Disponível em: ⟨https://doi.org/10.1145\%2F982962.964011⟩.

KLEIN, C.; CLEMENTS, J.; DIMOULAS, C.; EASTLUND, C.; FELLEISEN, M.; FLATT, M.; MCCARTHY, J. A.; RAFKIND, J.; TOBIN-HOCHSTADT, S.; FINDLER, R. B. Run your research. *ACM SIGPLAN Notices*, Association for Computing Machinery (ACM), 2012.

KRISHNASWAMI, N. R.; YALLOP, J. A typed, algebraic approach to parsing. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. [S.l.]: ACM, 2019.

KUAN, G.; MACQUEEN, D.; FINDLER, R. B. A rewriting semantics for type inference. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2007. p. 426–440. Disponível em: ⟨https://doi.org/10.1007\%2F978-3-540-71316-6_29⟩.

MEDEIROS, S.; MASCARENHAS, F.; IERUSALIMSCHY, R. Left recursion in parsing expression grammars. *Science of Computer Programming*, Elsevier BV, v. 96, p. 177–190, dec 2014. Disponível em: ⟨https://doi.org/10.1016\%2Fj.scico.2014.01.013⟩.

RIBEIRO, R.; REIS, L. V. S.; FEITOSA, S.; CARDOSO, E. M. Towards typed semantics for parsing expression grammars. In: *Proceedings of the XXIII Brazilian Symposium on Programming Languages*. ACM, 2019. Disponível em: ⟨https://doi.org/10.1145\%2F3355378.3355388⟩.