UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ARTUR TRÊS DO AMARAL

SimpleLambda: uma linguagem funcional didática com tradução para o Cubo Lambda

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Álvaro Freitas Moreira
Coorientador: Prof. Dr. Rodrigo Machado

Porto Alegre
2023

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Profª. Patricia Pranke
Pró-Reitora de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Lucas de Mello Schnorr
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

RESUMO

Cálculo Lambda é um modelo de computação proposto por Alonzo Church. É baseado na definição e aplicação de funções anônimas, sendo o comportamento dos termos definido por meio da substituição de parâmetros por argumentos. Em continuação desses estudos, Church também propõe uma variação com tipos, o Cálculo Lambda Simplesmente Tipado (STLC). Com a adição de um sistema de tipos, o cálculo perde expressividade e deixa de ser Turing Completo, mas adquire uma propriedade nova, a Normalização Forte. Todos os termos bem-tipados do STLC garantidamente chegam a uma forma normal, ou seja terminam a sua execução. Nas décadas seguintes, outros cálculos tipados foram desenvolvidos, adicionando funcionalidades como: polimorfismo, construtores de tipos e tipos dependentes. Essas funcionalidades aumentam a expressividade em relação ao STLC e mantêm a Normalização Forte. Posteriormente foi proposta uma estrutura de organização para relacionar esses cálculos, o Cubo Lambda. Partindo de STLC, cada aresta representa a adição de uma dessas funcionalidades, e cada vértice representa o cálculo resultante. Na disciplina de Tópicos Especiais em Cálculo Lambda e Teoria de Tipos do curso de Ciência da Computação da UFRGS, esses cálculos são vistos com o suporte de simuladores onde se pode definir e avaliar termos de cálculo lambda tipado. Esses simuladores foram desenvolvidos pelo professor Rodrigo Machado do Instituto de Informática da UFRGS, e oferecem diversas funcionalidades para a definição e visualização dos termos. Para cada cálculo visto na disciplina há um simulador diferente com uma sintaxe própria para expressar as funcionalidades do cálculo que está simulando, o que pode dificultar o processo de relacionar esses cálculos entre eles. Este trabalho propõe uma linguagem de programação funcional simples chamada SimpleLambda, e desenvolve uma aplicação web onde o código SimpleLambda pode ser compilado para os simuladores dos quatro cálculos vistos na disciplina de Cálculo Lambda e Teoria de Tipos.

Palavras-chave: Cálculo lambda tipado. compiladores. linguagens de programação. geração de código.

SimpleLambda: a didactic functional language with translation to the Lambda Cube

ABSTRACT

Lambda Calculus is a model of computation proposed by Alonzo Church. It is based on the definition and application of anonymous functions, with each term's behavior being defined by the substitution of parameters for arguments. Continuing these studies, Church also proposes a typed variation, Simply Typed Lambda Calculus (STLC). With the addition of a type system, the calculus loses expressive power and is no longer Turing Complete, it does however, acquire a new property: Strong Normalization. All well typed STLC terms are guaranteed to reach a normal form, that is, finish its execution. In the following decades, many other typed calculi were developed, adding functionalities such as: polymorphism, type constructors and dependent types. These functionalities increase the expressive power compared to STLC and retain Strong Normalization. A structure was later proposed to organize and draw relations between the calculi, it was named the Lambda Cube. Starting with STLC, each edge represents the addition of one of these functionalities, and each vertex represents the resulting calculus. In the Special Topics in Lambda Calculus and Type Theory class of the Computer Science curriculum at UFRGS, these calculi are studied with the support of online simulators where terms can be defined and evaluated. These simulators were developed by professor Rodrigo Machado of the Institute of Informatics of UFRGS, and they offer many functionalities to define and visualize lambda terms. There is a different simulator for each of the studied calculi, with its own syntax to express the functionalities of calculus being simulated, which can make it more difficult to draw relations between these calculi. This work proposes a simple functional programming language called SimpleLambda and implements a web application where SimpleLambda code can be compiled to the simulators of any of the four calculi studied in the Lambda Calculus and Type Theory class.

Keywords: typed lambda calculus. compilers. programming languages. code generation.

LISTA DE FIGURAS

Figura 4.1 Página web da ferramenta....25
Figura 4.2 Definição da AST em PureScript....27
Figura 4.3 Definição das configurações da linguagem SimpleLambda e geração de parsers para os tokens....28
Figura 4.4 Parser de IF....29
Figura 4.5 Parser de Termos (exceto operações ariméticas e booleanas, pares e aplicação)....29
Figura 4.6 Tabela de Operadores....30
Figura 4.7 Construindo um parser de expressões....30
Figura 4.8 Definição dos Tipos....31
Figura 4.9 Código para a regra de aplicação....31
Figura 4.10 Termos que representam subtração....41

LISTA DE ABREVIATURAS E SIGLAS

STLC Simply Typed Lambda Calculus

AST Abstract Syntax Tree

LISTA DE SÍMBOLOS

$\lambda_{\rightarrow}$	Cálculo Lambda Simplesmente Tipado
$\lambda 2$	Cálculo Lambda Polimórfico
$\lambda\omega$	Cálculo Lambda Polimórfico de Alta Ordem
λC	Cálculo de Construções

SUMÁRIO

1 INTRODUÇÃO 9
1.1 Motivação 9
1.2 Objetivo 10
1.3 Estrutura do Texto 11
2 BACKGROUND 12
2.1 Cálculo Lambda 12
2.2 Cálculo Lambda Simplesmente Tipado 14
2.3 Cubo Lambda 15
2.4 Cálculo Lambda Polimórfico ($\lambda 2$) 17
2.5 Cálculo Lambda Polimórfico de Alta Ordem ($\lambda\omega$) 18
2.6 Cálculo de Construções (λC) 19
3 A LINGUAGEM 20
3.1 Definição de SimpleLambda 20
3.2 Recursão 21
3.3 Tipos e Açúcar Sintático 22
3.4 Sistema de Tipos 23
4 A FERRAMENTA 25
4.1 Uso da Ferramenta 25
4.2 Arquitetura 26
4.2.1 Parsing 26
4.2.2 Inferência de Tipos 31
4.2.3 Geração de Código 32
4.2.4 Cálculo Lambda Simplesmente Tipado (STLC, $\lambda_{\rightarrow}$) 32
4.2.5 Cálculo Lambda Polimórfico ($\lambda 2$) 39
4.2.6 Cálculo Lambda Polimórfico de Alta Ordem ($\lambda\omega$) 42
4.2.7 Cálculo de Construções (λC) 43
4.3 Tecnologias usadas 44
5 CONCLUSÃO 45
REFERÊNCIAS 47

1 INTRODUÇÃO

Cálculo Lambda (CHURCH, 1932) é um modelo de computação proposto por Alonzo Church durante a década de 1930. Ele é baseado em definição e aplicação de funções anônimas, sendo o comportamento dos termos definido por meio da substituição de parâmetros por argumentos.

Em continuação do desenvolvimento de Cálculo Lambda, Church também propõe uma variação com tipos, o Cálculo Lambda Simplesmente Tipado (STLC) (CHURCH, 1940). Ao criar uma abstração lambda, além do nome da variável, é informado qual tipo que essa variável deve ter. Caso tente-se aplicar um termo que não corresponde ao tipo esperado, o resultado é um termo mal tipado. Com essa restrição, muitos termos importantes para a programação em Cálculo Lambda não podem ser bem tipados, principalmente os termos que usam auto-aplicação, usados para implementar recursão geral (Seção: 2.1). Com essa disciplina de tipos, o cálculo perde expressividade e deixa de ser Turing Completo, e adquire uma propriedade que não tinha antes, a Normalização Forte: todos os termos bem tipados do STLC garantidamente chegam a uma forma normal, ou seja terminam a sua execução.

Ao longo do século 20, muitas outras variações do cálculo lambda tipado foram desenvolvidas, tais como Cálculo Lambda Polimórfico (REYNOLDS, 1974) (ou system F (GIRARD, 1971)), Cálculo Lambda Tipado com Construtores de Tipos (GIRARD, 1972) e Cálculo Lambda Tipado com Tipos Dependentes (COQUAND; HUET, 1988). Essas funcionalidades aumentam a expressividade em relação ao STLC e mantêm a Normalização Forte. Esses cálculos são muito úteis para a lógica, pois é possível codificar provas lógicas usando termos de cálculo lambda tipado. Essa relação entre programas e provas, e entre tipos e proposições, é chamada Isomorfismo de Curry-Howard (HOWARD, 1980), que é a base do funcionamento de muitos assistentes de prova.

1.1 Motivação

Um possível obstáculo ao estudo desses sistemas é a complexidade da notação usada para representar esses termos lambda. Apesar de serem proximamente relacionados e partirem da mesma base, a sintaxe usada para representar as funcionalidades de cada cálculo pode variar bastante.

Na disciplina de Tópicos Especiais em Cálculo Lambda e Teoria de Tipos do curso

de Ciência da Computação da UFRGS, esses cálculos são vistos com o suporte de simuladores com os quais se pode definir e avaliar termos de cálculo lambda tipado. Esses simuladores foram desenvolvidos pelo professor Rodrigo Machado do Instituto de Informática da UFRGS, e oferecem diversas funcionalidades como a visualização das árvores de sintaxe abstrata dos termos, inferência de tipos e algum açúcar sintático para definição de variáveis, termos e tipos. Com isso uma maneira de programar nesses simuladores é montar uma biblioteca de termos básicos como definições e, a partir disso, formar termos mais complexos.

Para cada cálculo visto na disciplina de Tópicos há um simulador diferente, e cada um precisa de uma sintaxe diferente para expressar as funcionalidades do cálculo que está simulando. Essas diferenças podem fazer com que seja mais difícil para um aluno relacionar esses cálculos entre si.

Por motivos didáticos, seria interessante ter uma maneira de traduzir um mesmo programa escrito em uma linguagem de mais alto nível em mais de um cálculo, e comparar como cada um expressa esse programa.

1.2 Objetivo

Este trabalho se propõe a desenvolver uma ferramenta didática que facilite a programação em diferentes sistemas de cálculo lambda tipado e ajude a visualizar as diferenças entre eles. O usuário deve poder descrever um programa em uma linguagem de mais alto nível, similar ao núcleo de linguagens funcionais tais como OCaml e Haskell, e a partir disso, gerar código para qualquer um dos quatro cálculos vistos na disciplina de Cálculo Lambda e Teoria de Tipos.

A ferramenta desenvolvida neste trabalho é voltada para a programação nesses cálculos. É proposta uma linguagem de programação funcional simples chamada SimpleLambda, e uma aplicação web onde o código SimpleLambda pode ser compilado para quatro variações de cálculo lambda tipado.

A ferramenta está disponível em: <https://www.inf.ufrgs.br/~atamaral/simpleLambda/>

1.3 Estrutura do Texto

Este trabalho é organizado da seguinte maneira: O Capítulo 2 faz uma revisão dos conceitos teóricos relevantes a este trabalho. O Capítulo 3 descreve a linguagem SimpleLambda. O Capítulo 4 descreve a ferramenta desenvolvida, e o processo de compilar SimpleLambda para cada um dos cálculos tipados. O Capítulo 6 discute as conclusões sobre o trabalho e trabalhos futuros.

2 BACKGROUND

Neste capítulo são revisados os principais conceitos teóricos sobre os quais este trabalho é baseado. É explicado e exemplificado o funcionamento dos quatro cálculos que serão os alvos de compilação da ferramenta final, além das origens e a motivação para cada um deles.

2.1 Cálculo Lambda

Cálculo Lambda é um modelo formal de computação baseado na definição e aplicação de funções introduzido por Alonzo Church (CHURCH, 1932). Um termo lambda pode ser uma variável, uma função, ou uma aplicação de dois termos. As funções são definidas usando a notação λ para abstrair variáveis, por exemplo, em $\lambda x.x$, a variável imediatamente depois do λ indica que todas as ocorrências de x a seguir são ligadas ao lambda. A aplicação é representada por dois termos justapostos, aplicando o segundo como parâmetro para o primeiro. A sintaxe abstrata de termos do Cálculo Lambda é dada pela seguinte gramática:

$$M \quad ::= \quad x \mid \lambda x.\, M \mid MN$$

No caso de um termo na forma $(\lambda x.\ M)\ N$, temos uma expressão redutível, ou *redex*. Um *redex* é avaliado substituindo todas as ocorrências da variável abstraída x em M pelo termo N, resultando em $[N/x]\ M$.

Esse processo de substituir as ocorrências livres de x em M por N é chamado de redução beta ($\rightarrow_\beta$), e é o passo de execução para avaliar termos lambda. Um termo lambda que não contém nenhum *redex* é chamado de forma normal, e representa o resultado da avaliação de um termo. Por exemplo, abaixo temos um *redex* no lado esquerdo e uma forma normal no lado direito.

$$(\lambda x.\ x\ \ x)\ \ a \qquad \rightarrow_\beta \qquad a\ \ a$$

Com essa sintaxe, podemos codificar estruturas comuns para a computação, como numerais e booleanos. Para codificar os numerais, é usada a quantidade de aplicações de um termo sobre outro termo.

$$
\begin{aligned}
0 &= \lambda f.\ \lambda x.\ x \\
1 &= \lambda f.\ \lambda x.\ f\ x \\
2 &= \lambda f.\ \lambda x.\ f\ (f\ x) \\
3 &= \lambda f.\ \lambda x.\ f\ (f\ (f\ x)) \\
&\ldots
\end{aligned}
$$

Para representar booleanos, temos que levar em conta os construtores (*true* e *false*) e o destrutor (*if*). A representação de *true* e *false* é feita com o funcionamento de if em mente.

$$
\begin{aligned}
true &= \lambda a.\ \lambda b.\ a \\
false &= \lambda a.\ \lambda b.\ b
\end{aligned}
$$

$$
if = \lambda b.\ \lambda e_1.\ \lambda e_2.\ b\ e_1\ e_2
$$

Um *if* serve para decidir entre dois termos dependendo de um booleano. Caso seja *true*, o termo retornado é o primeiro, e caso seja *false*, o termo retornado é o segundo. Esse comportamento pode ser codificado diretamente nos construtores: *true* recebe dois termos e retorna o primeiro, e *false* recebe dois termos e retorna o segundo. Com essas codificações de *true* e *false*, o termo *if* precisa apenas receber um booleano e dois termos, aplicando os termos sobre o booleano.

Assim como o próprio cálculo, essas codificações para numerais e booleanos foram propostas por Alonzo Church, e por isso são chamadas respectivamente de numerais de Church e booleanos de Church.

Em Cálculo Lambda é possível formular termos lambda que nunca chegam a uma forma normal, mesmo após infinitas reduções. Por exemplo:

$$
(\lambda x.\ x\ \ x)\ \ (\lambda x.\ x\ \ x)
$$

Ao reduzir o *redex*, o termo resultante é exatamente igual ao termo inicial, então nenhum progresso é feito, como mostra a sequência de passos abaixo:

$$
(\lambda x.\ x\ \ x)\ \ (\lambda x.\ x\ \ x)\ \rightarrow_\beta\ (\lambda x.\ x\ \ x)\ \ (\lambda x.\ x\ \ x)\ \rightarrow_\beta\ (\lambda x.\ x\ \ x)\ \ (\lambda x.\ x\ \ x)\ \rightarrow_\beta\ \ldots
$$

Esse termo por si próprio não é muito útil para representar programas, mas com algumas modificações, podemos usar a ideia de termos autorreplicantes para representar funções recursivas. O termo lambda mais conhecido para representar recursão é o combinador Y, que recebe um termo e o replica infinitas vezes.

$$Y \;=\; \lambda f.\,(\lambda x.\,f\;(x\;\;x))\;\;(\lambda x.\,f\;(x\;\;x))$$

Se colocarmos um condicional nesse termo, podemos criar um ponto de parada para essa expansão do combinador Y. Supondo que já tenhamos as codificações do teste de zero *isZero* e das operações de decremento *dec* e de multiplicação *mult*, uma maneira de definir fatorial em cálculo lambda seria:

$$\begin{array}{lcl} Y & = & \lambda f.\,(\lambda x.\,f\;(x\;\;x))\;\;(\lambda x.\,f\;(x\;\;x)) \\ F & = & \lambda R.\,\lambda n.\;\; if \;\;\; (isZero\;(dec\;n)) \;\;\; 1 \;\;\; (mult\;n\;(R\;(dec\;n))) \end{array}$$

$$fatorial \;=\; Y\;\;F$$

Para realizar a computação do fatorial, multiplicamos o valor de entrada pelo seu valor decrementado em 1, sucessivas vezes até chegar em 1, e então chegamos ao resultado.

2.2 Cálculo Lambda Simplesmente Tipado

Em 1940, Alonzo Church introduziu o Cálculo Lambda Simplesmente Tipado (STLC) (CHURCH, 1940), que adiciona tipos ao Cálculo Lambda. Isso significa que ao definir uma função, deve-se informar qual é o tipo esperado para a variável ligada. A sintaxe abstrata de STLC é dada pela seguinte gramática:

$$\begin{array}{lcl} M & ::= & x \;\mid\; \lambda x{:}\,T.\,M \;\mid\; MN \\ T & ::= & X \;\mid\; T_1 \;\to\; T_2 \end{array}$$

Um tipo em STLC pode ser um tipo atômico representado por X na gramática acima, ou um tipo função representado por $T_1 \;\to\; T_2$. Por exemplo, o termo a seguir só é bem-tipado se N for um termo bem-tipado do tipo T.

$$(\lambda x : T.\,M) \;\;\; N$$

Para trazer os naturais e booleanos de Church para STLC, vamos ter que encontrar tipos apropriados. O booleano *true* de Church é codificado pelo seguinte termo no STLC

$$\lambda a :\; A.\,\lambda b :\; A.\,a \;\;:\;\; A \to A \to A$$

Portanto o tipo dos booleanos em STLC é $A \to A \to A$. No entanto *true* pode ser codificado com outro tipo no lugar de A, como por exemplo:

$$\lambda a :\ B.\ \lambda b :\ B.\ a \quad : \quad B \to B \to B$$

Assim, temos infinitas maneiras de atribuir tipos aos booleanos, todas igualmente válidas. O mesmo vale para as definições de *false* e *if*:

$$\begin{aligned} true &= \lambda a : A.\ \lambda b : A.\ a \\ false &= \lambda a : A.\ \lambda b : A.\ b \end{aligned}$$

$$if = \lambda b : (A \to A \to A).\ \lambda e1 : A.\ \lambda e2 : A.\ b\ \ e1\ \ e2$$

Podemos também atribuir tipos aos numerais de Church:

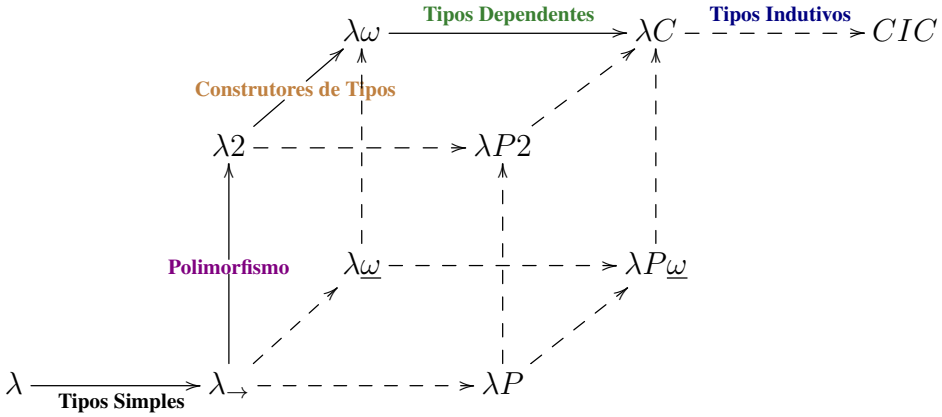
$$\begin{aligned} 0 &= \lambda f : (A \to A).\ \lambda x : A.\ x \\ 1 &= \lambda f : (A \to A).\ \lambda x : A.\ f\ x \\ 2 &= \lambda f : (A \to A).\ \lambda x : A.\ f\ (f\ x) \\ 3 &= \lambda f : (A \to A).\ \lambda x : A.\ f\ (f\ (f\ x)) \\ &... \end{aligned}$$

Assim como os booleanos, a codificação acima dos naturais pode usar qualquer tipo válido em STLC.

Em STLC não é possível atribuir tipos para o termo de auto-aplicação. Para um termo x_1 receber x_2 como parâmetro, o tipo de x_1 deve ser $T \to T'$, sendo T o tipo de x_2. Um único termo não pode ter esses dois tipos ao mesmo tempo. Portanto, a auto-aplicação não é válida em STLC. Com isso, também não temos como atribuir tipos ao combinador Y.

2.3 Cubo Lambda

A partir de STLC, muitas outras versões de Cálculo Lambda tipado foram desenvolvidas, estendendo a sua expressividade com novas funcionalidades. Então, em 1991 Henk Barendregt propôs a estrutura de um Cubo Lambda (BARENDREGT, 1991) para organizar esses cálculos, tendo como "ponto de partida"o STLC, representado na figura abaixo pelo vértice $\lambda_\to$ no canto inferior esquerdo.



Fonte: Slides da Disciplina de Cálculo Lambda e Teoria de Tipos

Cada eixo do cubo representa uma funcionalidade:

- Polimorfismo *(Termos que dependem de Tipos)*
- Construtores de Tipo *(Tipos que dependem de Tipos)*
- Tipos dependentes *(Tipos que dependem de Termos)*

Cada um dos cantos do cubo representa um cálculo com uma combinação de funcionalidades:

Cálculo	Polimorfismo	Construtores de Tipo	Tipos dependentes
$\boldsymbol{\lambda_{\rightarrow}}$	×	×	×
$\boldsymbol{\lambda 2}$	✓	×	×
$\lambda\underline{\omega}$	×	✓	×
λP	×	×	✓
$\boldsymbol{\lambda\omega}$	✓	✓	×
$\lambda P2$	✓	×	✓
$\lambda P\underline{\omega}$	×	✓	✓
$\boldsymbol{\lambda C}$	✓	✓	✓

Os cálculos relevantes para este trabalho são: $\lambda_{\rightarrow}$, $\lambda 2$, $\lambda\omega$, λC. Esses são os alvos de compilação para a ferramenta desenvolvida, que veremos em mais detalhes nas próximas seções.

2.4 Cálculo Lambda Polimórfico ($\lambda 2$)

Em $\lambda 2$, é adicionado polimorfismo a STLC, então agora podemos criar abstrações para tipos sobre termos. Assim como λ era usado para abstrair termos, podemos usar Λ para abstrair tipos na definição de um termo. A sintaxe abstrata de termos de $\lambda 2$ é dada pela seguinte gramática:

$$\begin{aligned} M \quad ::= \quad & x \mid \lambda x : T.\, M \mid MN \\ & \mid \Lambda X.M \quad (abstrai\ tipos) \\ & \mid M\ [\,T\,] \quad (aplica\ tipos) \end{aligned}$$

Como podemos abstrair os tipos dentro de um termo, o tipo desse termo deve indicar isso. Isso é feito com a sintaxe de $\forall$ que indica que o tipo abstraído pode ser substituído por qualquer tipo.

$$T \quad ::= \quad X \mid T_1 \to T_2 \mid \forall X.T$$

Usando abstrações de tipos, podemos definir booleanos genéricos.

$$\begin{aligned} true \quad &= \quad \Lambda C.\ \ \lambda a : C.\ \ \lambda b : C.\ \ a \quad : \quad \forall\, C.\, C \to C \to C \\ false \quad &= \quad \Lambda C.\ \ \lambda a : C.\ \ \lambda b : C.\ \ b \quad : \quad \forall\, C.\, C \to C \to C \end{aligned}$$

Ao aplicar um tipo T ao termo de um booleano, o tipo dos booleanos, $\forall C.C \to C \to C$, será instanciado para o tipo $T \to T \to T$, e o booleano se tornará um seletor de termos do tipo T. Dessa forma o termo *if* é codificado como segue, onde D representa o tipo da saída do *if* (os tipos da parte "*then*"e "*else"*).

$$if \quad = \quad \Lambda D.\ \ \lambda b : (\forall C.\, C \to C \to C).\quad \lambda e_1 : D.\quad \lambda e_2 : D.\quad b\ [D]\ e_1\ \ e_2$$

Os naturais também podem ser definidos de uma maneira genérica agora.

$$\begin{aligned} 0 \quad &= \quad \Lambda C.\ \ \lambda f : (C \to C).\, \lambda x : C.\, x \\ 1 \quad &= \quad \Lambda C.\ \ \lambda f : (C \to C).\, \lambda x : C.\, f\ x \\ 2 \quad &= \quad \Lambda C.\ \ \lambda f : (C \to C).\, \lambda x : C.\, f\ (f\ x) \\ 3 \quad &= \quad \Lambda C.\ \ \lambda f : (C \to C).\, \lambda x : C.\, f\ (f\ (f\ x)) \\ &\ldots \end{aligned}$$

Para fins didáticos podemos definir $Bool = \forall C.\ C \to C \to C$ e $Nat = \forall C.\ (C \to C) \to C \to c$. Assim $if\ [\ Nat\ ]\ true\ \ 0\ \ 1$, por exemplo, é um termo do tipo *Nat* que resulta em 0.

Uma versão mais restrita do polimorfismo presente em $\lambda 2$ é bastante disseminada em linguagens de programação. Essa versão mais restrita é conhecida como polimorfismo let ou ainda polimorfismo de Hindley-Milner.

2.5 Cálculo Lambda Polimórfico de Alta Ordem ($\lambda\omega$)

Em $\lambda\omega$, são introduzidos construtores de tipos, ou seja tipos que dependem de tipos. A sintaxe abstrata de termos de $\lambda\omega$ é dada pela seguinte gramática:

$$\begin{aligned} M \ ::= \ & x \mid \lambda x : T.\ M \mid MN \\ & \mid \Lambda X{:}\ *.M \\ & \mid M\ [\,T\,] \end{aligned}$$

A única alteração à sintaxe de termos em relação a $\lambda 2$ é a inclusão de uma anotação de *kind* para as variáveis de tipos. Informalmente, os *kinds* são tipos de tipos. Formalmente os *kinds* são definidos da seguinte maneira:

$$k \ ::= \ * \mid k \to k$$

Um tipo do *kind* $*$, é dito um tipo completo, enquanto um tipo de algum *kind* no formato $k \to k$ é um construtor de tipo. Para formar esses construtores de tipo, temos agora uma maneira de abstrair tipos dentro de uma definição de tipos, como podemos ver na sintaxe dos tipos de $\lambda\omega$:

$$\begin{aligned} T \ ::= \ & X \mid T_1 \to T_2 \mid \forall X{:}\ *.T \\ & \mid \lambda X : k.T \\ & \mid T_1\ T_2 \end{aligned}$$

Para construtores de tipos também usamos λ, porém como informamos o *kind* da variável de tipo, não há confusão com abstrações de termos. Para exemplificar o funcionamento de construtores de tipo, vamos definir um construtor para tipos de função $T \to T$.

$$F \;=\; \lambda X : *.\; X \;\rightarrow\; X$$

O construtor F é do *kind*: $* \;\rightarrow\; *$. Supondo que $A : *$, ou seja que A é um tipo completo, a aplicação $F \;\; A$ retorna um tipo completo $A \rightarrow A$. No exemplo a seguir, usamos o construtor F para definir o tipo da variável x, aplicando sobre ele o tipo completo A.

$$\lambda x : F\,A.x \;\; \rightarrow_\beta \;\; \lambda x : A \rightarrow A.x$$

2.6 Cálculo de Construções (λC)

Em λC são introduzidos tipos dependentes, que são tipos que dependem de termos. Os tipos em λC podem ser:

$$\begin{array}{lll} T \;::=\; & X \mid T_1 \rightarrow T_2 \mid \forall X : *.T & \\ & \mid \lambda X : k.T \mid T_1 \;\; T_2 & \\ & \mid \Lambda x : T_1.T_2 & \text{(tipo que recebe um termo)} \\ & \mid T \;\; M & \text{(aplicação de termo sobre um tipo)} \end{array}$$

Criamos uma abstração da variável x que recebe um termo, e retorna um tipo. Com essa adição, a barreira que separa termos e tipos se torna menos clara.

Usando o isomorfismo de Curry-Howard, podemos codificar provas para a Lógica de Predicados de Alta Ordem.

Tipos dependentes não são usados na maioria das linguagens de programação, porém é muito importante para assistentes de prova, e λC é a base para o assistente Coq. Na ferramenta desenvolvida para este trabalho, os tipos dependentes não são usados na geração de código. É gerado código para λC, porém nenhuma construção da linguagem SimpleLambda faz uso dos tipos dependentes.

3 A LINGUAGEM

Como ponto de partida para a programação nos quatro cálculos, foi desenvolvida a linguagem SimpleLambda como base, que pode ser compilada para STLC, $\lambda 2$, $\lambda\omega$ e λC. Por ser uma linguagem simples que compila para versões tipadas do Cálculo Lambda, essa linguagem foi nomeada SimpleLambda. Esse nome foi inspirado no nome de PureScript, a linguagem usada para desenvolver o compilador, que é uma linguagem puramente funcional que compila para JavaScript.

A linguagem SimpleLambda é fortemente baseada em L1, que é uma adaptação da linguagem PCF (PLOTKIN, 1977), que por sua vez, é o core mais básico para o estudo de sistemas de tipos e semântica formal de linguagens de programação. L1 é usada na disciplina de Semântica Formal do curso de Ciência da Computação da UFRGS para demonstrar conceitos como sistemas de tipos e diferentes formas de definir semântica operacional para linguagens de programação. Como a disciplina de Semântica é obrigatória no currículo de Ciência da Computação da UFRGS, boa parte dos alunos que cursam a disciplina de Cálculo Lambda e Teoria de Tipos já estão familiarizados com a linguagem L1. L1 por sua vez tem muito em comum com o núcleo de linguagens de programação funcionais tais como, OCaml e Haskell.

3.1 Definição de SimpleLambda

Nesta seção, é apresentada a sintaxe abstrata da linguagem SimpleLambda assim como as regras do seu sistema de tipos, seguido de exemplos de programas escritos em SimpleLambda. Começamos pela definição da sintaxe abstrata da linguagem dada pela gramática abaixo:

$$
\begin{array}{rcl}
e & \in & \text{SimpleLambda} \\
e & ::= & b \mid n \mid e_1 \; op \; e_2 \mid \; ! \; e_1 \mid if \; e_1 \; then \; e_2 \; else \; e_3 \\
 & \mid & x \mid e_1 \; e_2 \mid func \; (x : T) \; \Rightarrow \; e \mid let \; x : T = e_1 \; in \; e_2 \\
 & \mid & (e_1, e_2) \mid fst \; e \mid snd \; e \\
 & \mid & natRec \; (e_1; \; e_2; \; e_3) \\
T & \in & Types \\
T & ::= & Nat \mid Bool \mid T_1 \; \rightarrow \; T_2 \mid T_1 \times T_2
\end{array}
$$

onde

$$
\begin{array}{lcll}
n & \in & \{0,1,2,3,4,...\} & (números\ naturais) \\
b & \in & \{true,\ false\} & (booleanos) \\
x & \in & Ident & (conjunto\ de\ identificadores) \\
op & \in & \{+,\ -,\ *,\ \&\&,\ ||,\ ==,\ !=,\ <,\ >\} & (operações\ aritméticas\ e\ relacionais \\
 & & & e\ conectivos\ lógicos\ and\ e\ or)
\end{array}
$$

3.2 Recursão

A principal diferença de SimpleLambda em relação a L1, é a ausência de funções recursivas definidas através de expressões `let rec`. Isso acontece porque nenhum dos cálculos alvo da ferramenta é capaz de representar recursão geral. Por eles terem a propriedade de normalização forte, todos os termos bem tipados desses cálculos necessariamente chegam a uma forma normal e terminam a execução.

Apesar disso, em SimpleLambda é possível, usando a expressão `natRec`, definir várias funções comumente escritas recursivamente, como fatorial e sequência de Fibonacci. Para uma melhor compreensão da expressão `natRec` vamos voltar à definição usual da função fatorial dada abaixo:

```
fatorial n = if (n == 0)
              1
             (n * (fatorial n-1))
```

Para um exemplo em que n = 5, ao final de todas as chamadas recursivas, temos um resultado da seguinte forma:

```
    5 * 4 * 3 * 2 * 1 * 1 = 120
```

Para calcular o fatorial de 5 com essa definição, foram feitas exatamente 5 chamadas recursivas da função fatorial, e quando chegamos ao zero, retornamos 1 sem fazer novas chamadas recursivas. Então uma alternativa sem usar recursão geral, seria definir um passo do cálculo de fatorial, e o executar n vezes sobre algum ponto de partida. Esse é o funcionamento de `natRec`, que não é tão conveniente, ou tão expressivo como recursão geral, mas é suficiente para várias funções importantes. Vamos ver a seguir uma

maneira de calcular fatorial em SimpleLambda.

```
let fact_base : Nat X Nat = (1,1) in

let fact_step : (Nat X Nat) -> (Nat X Nat) =
func (p: Nat X Nat) => ((fst p)+1, (fst p) * (snd p)) in

let factorial: Nat -> Nat =
func (n:Nat) => snd (natRec (n ; fact_step ; fact_base)) in

factorial 5
```

Listing 3.1 – Fatorial em SimpleLambda

Uma dificuldade de definir um passo de fatorial é que a cada passo precisamos multiplicar um número diferente. Para isso podemos usar um par como ponto de partida, em que um valor é um contador que guarda qual número devemos multiplicar a cada passo e o outro valor do par serve como um acumulador para o resultado final. A cada passo realizamos a multiplicação do contador pelo acumulador, e incrementamos o contador para o próximo passo. Usando `(1,1)` como ponto de partida (base), ao final de n passos o par resultante será `(n+1, n!)` e podemos extrair o resultado da segunda posição. No exemplo a seguir podemos ver passo a passo como as aplicações sucessivas chegam ao resultado esperado.

```
snd  (fact_step (fact_step (fact_step (fact_step (fact_step (1,1))))))
snd  (fact_step (fact_step (fact_step (fact_step (2,1)))))
snd  (fact_step (fact_step (fact_step (3,2))))
snd  (fact_step (fact_step (4,6)))
snd  (fact_step (5, 24))
snd  (6, 120)
120
```

3.3 Tipos e Açúcar Sintático

Em L1, todo o conjunto dos números inteiros são suportados, enquanto em SimpleLambda apenas os naturais fazem parte da linguagem. Existem maneiras de representar inteiros em cálculo lambda tipado, como por exemplo, codificar um número como um par de um booleano e um natural, de forma que o booleano representaria o sinal e o natural representaria o valor do número. No entanto, essa construção tornaria o código

gerado muito menos legível, pois requer a inclusão da codificação de pares em todos os números e operações aritméticas. Por isso, foram usados na ferramenta apenas naturais representados por numerais de Church.

Para programar em cálculo lambda tipado, pares são muito úteis, e inclusive servem para construir outras operações básicas da linguagem como subtração e todos os comparadores. Então ter pares como uma funcionalidade nativa em SimpleLambda facilitou bastante a geração de código dessas operações mais complexas (Seção: 4.2.5). Além de operações básicas da linguagem, pares também podem ser usados pelo usuário para definir funções como fatorial (3.1), sequência de Fibonacci, exponenciais e muitos outros. Por isso foi decidido incluir Pares como uma funcionalidade nativa de SimpleLambda.

Na sintaxe concreta adotada no simulador para SimpleLambda, a definição de funções anônimas permite a introdução de mais de uma variável usando a sintaxe $func\ (x_1 : T_1) \ldots (x_n : T_n) \Rightarrow e$. Isso é um açúcar sintático para facilitar a programação. No exemplo abaixo, a remoção do açúcar sintático do programa (1) produz o programa (2).

```
(1) func (a:Nat) (b:Nat) (c:Nat) => a + b + c

(2) func (a:Nat)=> (func (b:Nat)=> (func (c:Nat)=> a + b + c))
```

3.4 Sistema de Tipos

Um sistema de tipos é um conjunto de regras de tipo que permitem atribuir tipos a termos de uma linguagem. Esse conjunto de regras define uma relação ternária entre um ambiente de variáveis (representado por Γ), um termo e e um tipo T, sendo $\Gamma \vdash e : T$ lido como e *é do tipo* T *sob o ambiente* Γ. Abaixo está a coleção de regras que define o sistema de tipos de SimpleLambda.

$$\frac{}{\Gamma \vdash n : Nat}\ (\text{T-NAT}) \qquad \frac{}{\Gamma \vdash b : Bool}\ (\text{T-BOOL}) \qquad \frac{\Gamma\,(x) = T}{\Gamma \vdash x : T}\ (\text{T-VAR})$$

$$\frac{\Gamma, x : T \vdash e : T'}{\Gamma \vdash func\,(x : T) \Rightarrow e : T \to T'}\ (\text{T-FUNC}) \qquad \frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ \ e_2 : T'}\ (\text{T-APP})$$

$$\frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash if\ \ e_1\ \ then\ \ e_2\ \ else\ \ e_3 : T} \quad \text{(T-IF)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1,\ e_2) : T_1\ \times\ T_2}\ \text{(T-PAIR)} \qquad \frac{\Gamma \vdash e : T_1\ \times\ T_2}{\Gamma \vdash fst\ \ e : T_1} \quad \text{(T-FST)}$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma, x : T \vdash e_2 : T'}{\Gamma \vdash let\ \ x : T = e_1\ \ in\ \ e_2 : T'}\ \text{(T-LET)} \qquad \frac{\Gamma \vdash e : T_1\ \times\ T_2}{\Gamma \vdash snd\ \ e : T_2} \quad \text{(T-SND)}$$

$$\frac{\Gamma \vdash e_1 : Nat \qquad \Gamma \vdash e_2 : T \to T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash natRec\ (e_1;\ e_2;\ e_3) : T} \quad \text{(T-NATREC)}$$

$$\frac{\Gamma \vdash e_1 : Nat \qquad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1\ \ opNat\ \ e_2 : Nat}\ \text{(T-OP-NAT)} \qquad \frac{\Gamma \vdash e_1 : Bool \qquad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1\ \ opBool\ \ e_2 : Bool}\ \text{(T-OP-BOOL)}$$

$$\frac{\Gamma \vdash e_1 : Nat \qquad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1\ \ opComp\ \ e_2 : Bool}\ \text{(T-OP-COMP)} \qquad \frac{\Gamma \vdash e_1 : Bool}{\Gamma \vdash\ \ !\, e_1 : Bool} \quad \text{(T-NOT)}$$

onde

$$\begin{aligned} opNat &\in \{+,\ -,\ *\} \\ opBool &\in \{\&\&,\ ||\} \\ opComp &\in \{==,\ !=,\ >,\ <\} \end{aligned}$$

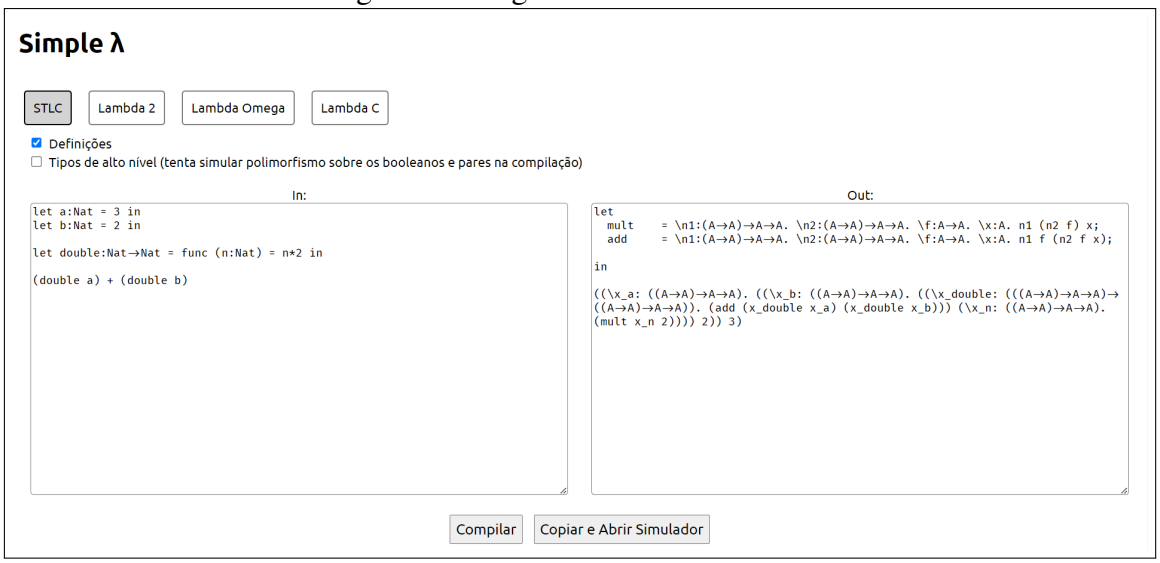
4 A FERRAMENTA

A ferramenta desenvolvida é uma aplicação web para a programação em SimpleLambda e geração de termos de cálculos lambda tipados do Cubo Lambda. A primeira seção desse capítulo descreve as características do layout da ferramenta e exemplos do seu uso. A segunda seção descreve a arquitetura da ferramenta, e é dividida em três partes: parsing, inferência de tipos e geração de código. A última seção desse capítulo lista as tecnologias usadas para implementar a ferramenta e por que elas foram escolhidas.

4.1 Uso da Ferramenta

A ferramenta (Figura 4.2) está disponível online em <https://www.inf.ufrgs.br/~atamaral/simpleLambda/>. Através dela, o usuário pode escrever um programa em SimpleLambda e compilar para um dos quatro cálculos alvo, STLC, $\lambda 2$, $\lambda\omega$ e λC.

Figura 4.1: Página web da ferramenta



Quatro botões possibilitam a escolha da linguagem alvo de compilação. Logo abaixo um checkbox define se a compilação vai usar o açúcar sintático de definição de termos disponível nos simuladores para as primitivas de SimpleLambda. Por exemplo, o Listing 4.1 mostra o resultado da compilação da expressão SimpleLambda `1 + 1` com a opção “Definições” selecionada e a Listing 4.2 mostra o resultado sem definições.

```
let
    add = \n1:(A->A)->A->A.
          \n2:(A->A)->A->A.
          \f:A->A. \x:A. n1 f (n2 f x);
in
(add 1 1)
```

Listing 4.1 – Código gerado com definições

```
((\n1:(A->A)->A->A.\n2:(A->A)->A->A.\f:A->A
    . \x:A. n1 f (n2 f x))(\f:A->A.\x:A.(f
    x)) (\f:A->A.\x:A.(f x)))
```

Listing 4.2 – Código gerado sem definições

No caso de STLC, uma opção extra está disponível para o uso de tipos de alto nível, que será explicada na seção sobre geração de código (Seção 4.2.4). No bloco de texto à esquerda da ferramenta, é feita a entrada de código em SimpleLambda e no bloco à direita é onde o código gerado é disposto. Dois botões se encontram imediatamente abaixo dos blocos de texto, o botão “Compilar” e o botão “Copiar e Abrir no Simulador”. O botão “compilar” faz a compilação do programa de entrada e coloca o termo resultante no bloco de texto de saída. O botão “Copiar e Abrir no Simulador” copia o texto do bloco de saída para a área de transferência e abre em outra aba a página do simulador correspondente ao cálculo alvo selecionado.

4.2 Arquitetura

O processo para transformar código SimpleLambda em cálculo lambda tipado é formado por três partes: parsing do texto de entrada para montar uma AST, verificação de tipos sobre a AST e geração de código a partir da AST.

4.2.1 Parsing

O processo de parsing de uma linguagem de programação tem como objetivo gerar, a partir de um programa, uma árvore sintática abstrata (AST) que represente a estrutura desse programa recebido. Essa AST então será a base para a inferência de tipos e para a geração de código.

A AST é definida como um *data type* chamado `Term` (Figura 4.2) com um construtor para cada estrutura sintática de SimpleLambda, conforme a gramática abstrata da

linguagem dada no Capítulo 3.

Nos construtores das ASTs para funções anônimas e para expressões `Let`, um dos parâmetros é um tipo, então é usado o *data type* `TermType` (Figura 4.8 na Seção 4.2.2 sobre Inferência de Tipos). Além disso são definidos os *data types* `BinopCode` para operações binárias, `UnopCode` para unárias e o *data type* `Ident` como um nome alternativo para `String`.

Figura 4.2: Definição da AST em PureScript

```
data BinopCode = Add | Sub | Mult | Lt | Gt | Eq | Ne | And | Or
data UnopCode = Not

data Term    = T_true
             | T_false
             | T_num Int
             | T_if Term Term Term
             | T_pair Term Term
             | T_fst Term
             | T_snd Term

             | T_binop BinopCode Term Term
             | T_unop UnopCode Term

             | T_natRec Term Term Term
             | T_var Ident
             | T_func Ident TermType Term
             | T_app Term Term
             | T_let Ident TermType Term Term

             | T_func_system Ident TermType Term
             | T_var_system Ident
```

Para implementar um parser de SimpleLambda, foi usada a biblioteca Parsing (PARSING, 2017) da linguagem PureScript, que é uma reimplementação da biblioteca Parsec(PARSEC, 2006) de Haskell. O seu funcionamento é baseado em combinadores de parser, o que nos permite definir vários parsers menores e combiná-los para formar um parser mais complexo.

A biblioteca também fornece módulos específicos para fazer o parsing de linguagens de programação (TOKEN, 2017), o que nos permite definir palavras reservadas, formato dos identificadores, operadores e até o formato dos comentários que a linguagem deve aceitar. A partir dessa definição, a função `makeTokenParser` gera parsers para todos os tokens (palavras reservadas, operadores, identficadores e etc) descritos na definição da linguagem em uma estrutura *record*. A Figura 4.3 mostra o trecho de código da

biblioteca Parsing de PureScript para definição das configurações da linguagem e geração de parsers para os tokens.

Figura 4.3: Definição das configurações da linguagem SimpleLambda e geração de parsers para os tokens

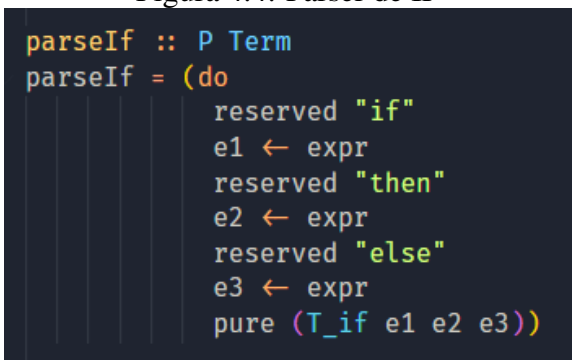
```
languageDef :: LanguageDef
languageDef = LanguageDef
  { commentStart: "{-"
  , commentEnd: "-}"
  , commentLine: "--"
  , nestedComments: false
  , identStart: lower
  , identLetter: alphaNum <|> (char '_')
  , opStart: oneOf ['-', ':', '+', '*', '|', '=', '~', '<', ' ', 'X']
  , opLetter: oneOf ['=', '|']
  , reservedNames: ["true", "false", "if", "then", "else", "fst", "snd",
  "let", "in", "func", "Nat", "Bool", "natRec"]
  , caseSensitive: true
  , reservedOpNames: ["=", "+", "-", "*", "||", "=", "~", "<", ":", "←", "X"]
}

token :: TokenParser
token = makeTokenParser languageDef
```

A partir do *record* `token`, podemos acessar todos os parsers de tokens gerados, por exemplo, `token.identifier` faz o parsing de identificadores, e `token.reserved` faz o parsing de uma palavra reservada recebida por parâmetro. A vantagem de usar os parsers de tokens gerados pela biblioteca Parsing é que eles já lidam com todo o espaço em branco, inclusive comentários. Então para criar um parser que identifica um token seguido de outro, não precisamos nos preocupar em consumir os espaços, tabs, quebras de linha, comentários, ou qualquer outro caractere classificado como espaço em branco que pode aparecer entre eles. Com esses parsers de tokens, podemos combiná-los para reconhecer todas as construções sintáticas da gramática abstrata de SimpleLambda.

Como um exemplo, na Figura 4.4, temos código da biblioteca Parsing de PureScript para expressão if-then-else de SimpleLambda:

Figura 4.4: Parser de IF

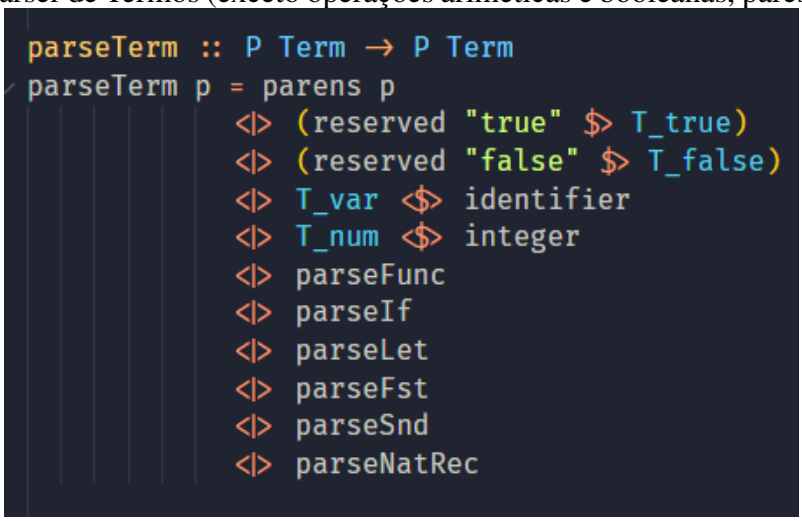


```
parseIf :: P Term
parseIf = (do
            reserved "if"
            e1 ← expr
            reserved "then"
            e2 ← expr
            reserved "else"
            e3 ← expr
            pure (T_if e1 e2 e3))
```

O tipo `P Term` (primeira linha da Figura 4.4), tem o mesmo significado que `Parser String Term`, o que significa que o parser recebe uma string e retorna uma AST, ou seja um valor do *data type* `Term` definido na Figura 4.2 (construído com `T_if`). Observe que o parser para expressões If da Figura 4.4 faz uso do parser `expr` que por sua vez é a combinação de todos os parsers da linguagem.

Usando parsers menores, como o `parseIF` da Figura 4.4, podemos criar um parser maior (Figura 4.5) que represente todos os formatos de termos que a linguagem oferece, exceto operações ariméticas e booleanas, pares e aplicação.

Figura 4.5: Parser de Termos (exceto operações ariméticas e booleanas, pares e aplicação)



```
parseTerm :: P Term → P Term
parseTerm p = parens p
            <|> (reserved "true" $> T_true)
            <|> (reserved "false" $> T_false)
            <|> T_var <$> identifier
            <|> T_num <$> integer
            <|> parseFunc
            <|> parseIf
            <|> parseLet
            <|> parseFst
            <|> parseSnd
            <|> parseNatRec
```

É preciso também fazer o parsing das operações aritméticas, lógicas e comparativas, além da aplicação de termos e a definição de pares. Isso é feito usando uma tabela de operadores do módulo Parsing.Expr (EXPR, 2017), como na Figura 4.6.

Figura 4.6: Tabela de Operadores

```
table :: OperatorTable Identity String Term
table =
  [ [ Prefix (reservedOp "!" $> T_unop Not)]
  , [ Infix (reservedOp "*"  $> T_binop Mult) AssocLeft]
  , [ Infix (reservedOp "+"  $> T_binop Add)  AssocLeft
    , Infix (reservedOp "-"  $> T_binop Sub)  AssocLeft]
  , [ Infix (reservedOp "<"  $> T_binop Lt)   AssocLeft
    , Infix (reservedOp ">"  $> T_binop Gt)   AssocLeft
    , Infix (reservedOp "=" $> T_binop Eq)   AssocLeft
    , Infix (reservedOp "≠" $> T_binop Ne)   AssocLeft
    , Infix (reservedOp "&&" $> T_binop And)  AssocLeft
    , Infix (reservedOp "||" $> T_binop Or)   AssocLeft]
  , [ Infix (whiteSpace      $> T_app)        AssocLeft]
  , [ Infix (reservedOp ","  $> T_pair)       AssocLeft]
  ]
```

Usando os colchetes, podemos definir grupos de precedência, por exemplo, o operador de multiplicação está acima de soma e subtração, que estão acima de todos os comparadores. Para definir a aplicação de termos, o espaço em branco é tratado como um operador. Nota-se que a definição de pares são dois termos separados por uma vírgula, sendo tecnicamente possível definir pares sem parenteses ao redor, porém isso não é recomendado.

Figura 4.7: Construindo um parser de expressões

```
expr :: P Term
expr = fix allExprs
  where
    allExprs p = buildExprParser table (parseTerm p)
```

Com a definição de termo e a tabela de operadores, podemos combinar para um parser único usando `buildExprParser`. A função `fix` de Purescript define esse parser como ponto fixo da função e temos o parser `expr` final que transforma o código de entrada em AST.

Um processo idêntico é feito para criar um parser dos tipos de SimpleLambda. O retorno desse parser é um `TermType` definido na seção sobre Inferência de Tipos (4.2.2).

4.2.2 Inferência de Tipos

Os tipos da linguagem são representados por um *data type* `TermType` definido na Figura 4.8 que define um construtor de tipo para cada tipo da linguagem SimpleLambda. Os tipos básicos são Nat e Bool e os construtores de tipos são Func e Pair.

Figura 4.8: Definição dos Tipos

```
data TermType    = Nat
                 | Bool
                 | Pair TermType TermType
                 | Func TermType TermType
```

A função `typeInfer` percorre a AST produzida pelo parser e retorna um tipo caso a AST seja bem tipada de acordo com as regras do sistema de tipos (vistas ao final do Capítulo 3). O resultado de `typeInfer` é do tipo `Maybe TermType`, então em caso de sucesso o resultado é `Just T`, sendo `T` o tipo inferido, e em caso de falha a função retorna `Nothing`. A função `typeInfer` segue estritamente o especificado pelas regras de tipo s. Como exemplo, abaixo segue a regra do sistema de tipos para aplicação e na Figura 4.9 o trecho da função `typeInfer` que implementa essa regra

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \;\; e_2 : T'} \qquad \text{(T-App)}$$

Figura 4.9: Código para a regra de aplicação

```
(T_app e1 e2) → (case typeInfer env e1 of
                   Just (Func t1 t2) → if (typeInfer env e2) == Just t1
                                       then Just t2
                                       else Nothing
                   _ → Nothing)
```

Com uma chamada recursiva para `TypeInfer` é verificado se e_1 é uma função de t_1 para t_2. Se for uma função, então é feita a inferência de tipo sobre e_2, e se o resultado for igual ao t_1 inferido em e_1, retorna t_2 como resultado. Se e_1 não for uma função, ou e_2 não corresponder com o tipo esperado por e_1, a função retorna `Nothing` indicando que a aplicação é mal-tipada.

4.2.3 Geração de Código

A partir de uma AST bem-tipada, pode-se gerar código de duas maneiras, com ou sem definições. O código gerado com definições tende a ser mais legível, porém depende do açúcar sintático disponível nos simuladores. Já a geração sem definições resulta em um termo único usando apenas a sintaxe do cálculo alvo escolhido, que dependendo da complexidade do programa original, pode ser praticamente ilegível. Na Seção 4.1 (Listing 4.1 e 4.2) há um exemplo das duas maneiras de gerar código para um programa simples.

Todos os simuladores dispõem de um bloco de definição de termos usando a sintaxe `let..in`, e todos menos STLC dispõem de um bloco de definição de tipos com a sintaxe `typedef..end`. Quando se seleciona a opção de usar definições, as definições dos tipos básicos de SimpleLambda são sempre incluídas, porém as definições das demais construções sintáticas de SimpleLambda são incluídas sob demanda, ou seja, apenas quando as construções aparecem no programa SimpleLambda de entrada.

Para saber quais definições devem ser incluídas no bloco gerado, acontece uma etapa extra de análise de dependências sobre a AST antes de gerar o código com definições. É gerada uma lista de todas as construções e operações usadas no termo original, e apenas essas têm as suas definições incluídas no bloco gerado. Isso é feito para ter o resultado mais compacto e mostrar apenas o que é necessário para representar o programa de entrada no cálculo alvo.

As definições do simulador são usadas apenas para representar primitivas da linguagem SimpleLambda, mas não são usadas para termos definidos pelo usuário. Isso acontece porque a declaração de variáveis por `let..in` de SimpleLambda serve também para definir o escopo em que essa variável pode ser acessada, e o bloco de definições do simulador tornaria a variável acessível em qualquer ponto do termo.

A opção de gerar código com definições pode ser usada com qualquer um dos cálculos alvo, exceto STLC com tipos de alto nível, devido ao grande número de definições que seriam necessárias e outras dificuldades detalhadas mais a seguir neste texto.

4.2.4 Cálculo Lambda Simplesmente Tipado (STLC, $\lambda_{\rightarrow}$)

STLC é o cálculo mais simples e mais limitado entre os quatro apresentados. Por causa de suas limitações, se mostrou um alvo de compilação inadequado para SimpleLambda. Essa inadequação fica evidente ao vermos a complexidade do processo de tra-

dução de SimpleLambda para STLC.

Um tipo T em STLC pode ser um tipo atômico (representados por um ou mais caracteres começando com letra maiúscula), ou um tipo $T_1 \rightarrow T_2$, onde T_1 e T_2 são tipos. Então para representar os tipos básicos de SimpleLambda (Nat e Bool), há uma quantidade infinita de tipos em STLC, uma para cada escolha dos tipos atômicos. Por exemplo, uma maneira válida de representar o tipo Bool seria `(A->A->A)`, porém seria igualmente válido usar `(B->B->B)`, ou usar qualquer outro tipo atômico como base. Por isso foi decidido que todos os termos STLC gerados na ferramenta serão construídos sobre o tipo arbitrário `A`.

Booleanos

Inicialmente a geração de código para STLC seria feita a partir de de uma biblioteca fixa de termos, apenas substituindo os nós da AST pelo equivalente em STLC. No entanto, dessa forma, um grande número de programas SimpleLambda resultaria em termos STLC mal-tipados. Por exemplo a tradução da expressão SimpleLambda `if true then 1 else 2` baseada em codificações fixas para `if` e `true`, levaria ao seguinte termo do STLC:

```
if true then 1 else 2

== Compila para =>

((\b:(A->A->A). \e1:A. \e2:A. b e1 e2)    -- if
(\a:A.\b:A.a)                             -- true
(\f:A->A.\x:A.(f x))                      -- 1
(\f:A->A.\x:A.(f (f x))))                 -- 2
```

O termo gerado é mal-tipado, e em primeira análise vemos que a definição de `if` espera um booleano `(A->A->A)` e dois termos do tipo `A`, mas além do booleano, ele recebe dois naturais do tipo `(A->A)->A->A`, o que torna o termo mal-tipado. Mesmo se ajustarmos a definição de `if` para receber naturais, o termo ainda seria mal-tipado, pois o funcionamento do `if` é usar o booleano como seletor dos outros dois parâmetros, retornando o primeiro se for verdadeiro e o segundo se for falso, e nesse caso o booleano funciona como um seletor para termos do tipo `A`, e ao receber naturais, o termo se torna mal-tipado.

Para resolver esse problema foi desenvolvida uma segunda opção para gerar STLC

com tipos de alto nível.

Para gerar um termo bem-tipado a partir desse termo SimpleLambda, é necessário alterar dinamicamente as definições de tipo do termo `if` e dos booleanos, dependendo do tipo de retorno do `if`. Ao inferir que o tipo dos parâmetros correspondentes as partes `then` e `else` do `if` são do tipo Nat, os booleanos usados devem se tornar seletores de naturais, ao invés de seletores de termos do tipo A, e da mesma forma, o `if` deve receber um seletor de naturais e dois naturais. Para chegar à anotação de tipo para booleanos seletores de naturais, cada `A` no tipo `A -> A -> A` é substituído pela representação de Nat em STLC `(A->A) -> A -> A` resultando em:

```
    (((A->A)->A->A) -> ((A->A)->A->A) -> ((A->A)->A->A))
```

De maneira análoga pode-se criar booleanos seletores de booleanos:

```
    ((A->A->A) -> (A->A->A) -> (A->A->A))
```

Fazendo os ajustes necessários ao `if` e ao `true`, já podemos gerar um termo bem tipado para a entrada original:

```
if true then 1 else 2

== Compila para =>

((\b: (((A->A)->A->A) -> ((A->A)->A->A) -> ((A->A)->A->A)).
  \e1: ((A->A)->A->A). \e2: ((A->A)->A->A). b e1 e2)        -- if
(\a:((A->A)->A->A).\b:((A->A)->A->A).a)                     -- true
(\f:A->A.\x:A.(f x))                                        -- 1
(\f:A->A.\x:A.(f (f x))))                                   -- 2
```

Por consequência dessas adaptações, o mesmo termo `true` pode ser traduzido para infinitos termos STLC diferentes dependendo do contexto em que eles se encontram e qual tipo que eles precisam selecionar. Isso também implica que as operações lógicas como And, Or e Not também devem ter suas anotações de tipo geradas dinamicamente para se adaptar à representação do tipo Bool sobre a qual devem operar.

Pares

Outra construção que precisa dessa abordagem são os pares, que assim como em `if`, têm a sua definição baseada em selecionar termos usando booleanos.

Em uma primeira versão da tradução de SimpleLambda para STLC, a definição de pares usada em STLC aceitava apenas pares de naturais, e tentava se adequar à definição padrão dos booleanos sobre tipos atômicos `(A->A->A)`. Isso era feito inserindo

os numerais em termos do tipo `A`, usando um seletor do tipo `A` e em seguida extraindo o natural desse termo. Ver o exemplo a seguir:

```
pair = \n1:(A->A)->A->A.        -- primeiro valor
       \n2:(A->A)->A->A.        -- segundo valor
       \b:(A->A->A).            -- booleano
          \f:A->A.
          \x:A.
          b (n1 f x) (n2 f x);

(pair 1 2) tem o tipo (A->A->A)->((A->A)->A->A)
                        Bool  ->    Nat
```

Depois do lambda que recebe o booleano, são criadas duas variáveis `f:A->A` e `x:A`. Essas variáveis são aplicadas aos naturais, que tem o tipo `(A->A)->A->A`, de forma que os termos `(n1 f x)` e `(n2 f x)` sejam do tipo `A`. Com esses termos do tipo `A`, podemos usar um booleano do tipo mais simples `A->A->A` (um seletor de termos do tipo A), para escolher qual valor será retirado do par. Depois que a seleção é feita, tudo que resta é um termo da forma `\f:A->A. \x:A. (n1 f x)`, como no exemplo no Listing 4.3, onde `n1 = 1`.

```
\f:A->A. \x:A. (\f:A->A. \x:A. f x)  f x

-- reduz para ->

\x:A. (\f:A->A. \x:A. f x)  x

-- reduz para ->

(\f:A->A. \x:A. f x)  -- numeral de Church usado para
                         representar o natural 1
```

Listing 4.3 – Resultado da aplicação sobre o booleano

Depois que as reduções são feitas nesse termo, o resulta final é apenas o natural 1, que foi retirado do termo `(n1 f x)`.

Essa codificação de pares funciona, e é uma solução elegante para combinar o tipo booleano mais simples da linguagem com a codificação dos naturais. Porém ela não nos

permite criar pares de booleanos, ou pares de pares, ou qualquer outro tipo de par, então precisamos de uma construção mais versátil.

A solução está novamente em estender os booleanos para que eles selecionem qualquer tipo. O termo que representa um par com dois valores deve ser uma função que recebe um seletor apropriado e retorna um dos valores. Chegamos então na seguinte anotação de tipo: `(T -> T -> T) -> T`, sendo `T` qualquer tipo inferido dos valores do par. Por exemplo um par (1,2) de SimpleLambda geraria o seguinte código STLC:

```
(\e1: ((A->A)->A->A).      -- primeiro valor
 \e2: ((A->A)->A->A).      -- segundo valor
 \b: (((A->A)->A->A) ->
      ((A->A)->A->A) ->
      ((A->A)->A->A)).     -- seletor de Nat
 b e1 e2)

(\f:A->A.\x:A.(f x))       -- 1
(\f:A->A.\x:A.(f (f x)))   -- 2
```

E para a definição de um destrutor de pares como o termo `fst`, criamos um termo que recebe o par do tipo `(Nat->Nat->Nat)->Nat`, e aplicaria sobre ele um termo `true` seletor de naturais para retornar um dos valores do par.

```
fst =
    \p: (((A->A)->A->A) ->
         ((A->A)->A->A) ->
         ((A->A)->A->A)) ->
          ((A->A)->A->A).
        p
        (\a:((A->A)->A->A).\b:((A->A)->A->A).a))
```

Um detalhe sobre a geração de pares para STLC, é que a ferramenta aceita apenas pares do mesmo tipo, como $Nat \times Nat$, $Bool \times Bool$, ou até $(Nat \to Nat) \times (Nat \to Nat)$, mas os dois termos do par devem ser do mesmo tipo. Pares de tipos diferentes podem ser definidos em SimpleLambda, porém esse caso não foi coberto na geração de código para STLC. Gerar código bem-tipado para STLC se mostrou bastante difícil, e por isso algumas funcionalidades acabaram sendo deixadas para trás.

Resumo Tipos de Alto Nível

Na geração dos termos com tipos de alto nível, deve se levar em conta o contexto, que pode ser diretamente inferido, ou recebido de uma inferência que ocorreu anteriormente. No caso o contexto seria um tipo `T` ao qual o termo atual deve se adaptar, que pode ser Nat, Bool ou qualquer combinação usando os operadores de Par e Função. E a partir desse tipo `T`, são usados os seguintes padrões de geração de código:

```
(=>) compila para
true  => \a:T. \b:T. a
false => \a:T. \b:T. b
if    => \b:T -> T -> T. \e1:T. \e2:T. b e1 e2

pair  => \e1:T. \e2:T. \b: T -> T -> T. b e1 e2
fst   => \p: (T -> T -> T) -> T. p (a:T. b:T. a)
snd   => \p: (T -> T -> T) -> T. p (a:T. b:T. b)

not   => \e1: (T -> T -> T). \a: T. \b: T. e1 b a
and   => \e1: (T -> T -> T).
         \e2: (T -> T -> T). \a: T. \b:T. e1 (e2 a b) b
or    => \e1: (T -> T -> T).
         \e2: (T -> T -> T). \a: T. \b:T. e1 a (e2 a b)
```

Custos dos tipos de alto nível

Essa abordagem de geração de código começa a mostrar os seus custos em casos de aninhamento de termos `if`. Podemos ver isso em um exemplo simples:

```
    if (if true then false else false) then 1 else 2
```

Listing 4.4 – Um nível de aninhamento

Esse termo é bem tipado em SimpleLambda, pois o `if` interno retorna um booleano. Porém pensando na tipagem dos termos, vemos que o `if` interno deve retornar um seletor de naturais para que possa ser usado para decidir entre o 1 e o 2. Mas para que esse `if` possa retornar um seletor de naturais, o termo `true` que seleciona os booleanos no `if` interno, deve ser tipado como um seletor de seletores de naturais, e portanto ter a seguinte assinatura de tipo:

```
    ((((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)) ->
```

```
      (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)) ->
      (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)))
```

E a complexidade pode aumentar ainda mais incluindo pares e aninhamentos mais profundos de termos if. Por exemplo um termo um pouco mais complexo com pares dentro do if interno:

```
if  fst (if true then (false,true) else (true, false)) then 1
   else 2
```

Se colocado na ferramenta, o termo resultante seria:

```
((\b: (((A->A)->A->A) -> ((A->A)->A->A) -> ((A->A)->A->A)). \e1: ((A->A)->A->A). \e2: ((A->A)->A->A). b e1 e2) ((\p:
     (((((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A))->(((A->A)->A->A)->((A->A)->A->A)->((A->A)->
     A->A)->((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A))). p (\a:(((A->A)->A
     ->A)->(((A->A)->A->A)->((A->A)->A->A))).\b:(((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))).a))((\b: ((((((A->
     A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->
     A->A)->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A)))) -> (((((A->A)->
     A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A
     )->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A)))) -> (((((A->A)->A->A
     )->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)
     ->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))))). \e1: (((((A->A)->A
     ->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)
     ->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A)))). \e2: (((((A->A)->A
     ->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)
     ->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A)))). b e1 e2) (\a:(((((A
     ->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)
     ->A->A)->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))))).\b:(((((A->A)
     ->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A
     ->A)->(((A->A)->A->A)->((A->A)->A->A)))) -> (((A->A)->A->A)->(((A->A)->A->A)->((A->A)->A->A))))).a) ((\e1: (((A->
     A)->A->A)->((A->A)->A->A)->((A->A)->A->A)). \e2: (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)). \b: ((((A->A)
     ->A->A)->((A->A)->A->A)->((A->A)->A->A)) -> (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A))) -> (((A->A)->A->A)
     ->((A->A)->A->A)->((A->A)->A->A))). b e1 e2) (\a:((A->A)->A->A).\b:((A->A)->A->A).b) (\a:((A->A)->A->A).\b:((A->
     A)->A->A).a)) ((\e1: (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)). \e2: (((A->A)->A->A)->((A->A)->A->A)->((A
     ->A)->A->A)). \b: ((((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A)) -> (((A->A)->A->A)->((A->A)->A->A)->((A->A)
     ->A->A)) -> (((A->A)->A->A)->((A->A)->A->A)->((A->A)->A->A))). b e1 e2) (\a:((A->A)->A->A).\b:((A->A)->A->A).a)
     (\a:((A->A)->A->A).\b:((A->A)->A->A).b)))) (\f:A->A.\x:A.(f x)) (\f:A->A.\x:A.(f (f x))))
```

Essa estratégia de tipagem dos termos é uma tentativa de replicar o funcionamento de uma linguagem com tipos polimórficos, mas podemos ver que é um processo trabalhoso e que gera termos muito extensos e difíceis de ler. Além disso, essa abordagem dificulta bastante a implementação, já que agora precisamos propagar a informação sobre o tipo de retorno e o contexto em que o booleano ou par se encontra, e usar essa informação para gerar os termos com os tipos corretos para o contexto.

Limitações de STLC

Mesmo usando tipos mais complexos não é possível representar operações como subtração e igualdade (HILLEBRAND; KANELLAKIS, 1996).

Como será visto na próxima seção (4.5), a estrutura dos naturais pode ser usada para fazer aplicações sucessivas de uma função sobre um ponto inicial, e isso é usado para traduzir `natRec` (visto na seção 3.2).

Para fazer isso em STLC, o tipo dos naturais teria que ser adaptado para comportar o tipo da função e do ponto de partida, assim como foi feito com os booleanos e os tipos de retorno do `if`. Infelizmente a ferramenta não faz essa adaptação, e todos os naturais gerados tem o tipo `((A->A)->A->A)`.

4.2.5 Cálculo Lambda Polimórfico ($\lambda 2$)

Em $\lambda 2$, temos a adição de tipos polimórficos, que significa que podemos definir tipos genéricos. Isso é feito com uma nova abstração de tipo representada por um lambda maiúsculo (Λ, ou no caso do simulador duas barras `\\`), para criar uma variável de tipo, e uma aplicação de tipo, na qual um termo recebe um tipo como argumento entre colchetes. Abaixo podemos ver como fica geração de código $\lambda 2$ para a expressão `if true then 1 else 2` de SimpleLambda.

```
  if true then 1 else 2

== Compila para =>

  ((\\D.\c:(forall C,C->C->C).\a:D.\b:D. (c[D]) a b)  -- if
  [(forall C, (C -> C) -> C -> C)]                    -- [Nat]
  (\\C.\a:C.\b:C.a)                                   -- true
  (\\C.\f:C->C.\x:C.(f x))                            -- 1
  (\\C.\f:C->C.\x:C.(f (f x))))                       -- 2
```

O `if` funciona com o mesmo mecanismo de aplicar as duas possibilidades ao valor booleano, porém vemos que a construção do booleano é diferente. O seu tipo é `(forall C, C->C->C)` que significa que um booleano (*true* ou *false*) funcionam como seletor de termos de qualquer tipo . Com isso temos uma maneira única de representar booleanos que podem se adaptar a qualquer contexto e selecionar qualquer tipo que for aplicado a eles.

Para casos de termos `if` aninhados, como no exemplo (4.4), o booleano que deve selecionar outros booleanos recebe o tipo Bool como parâmetro e retorna um booleano, que, por sua vez, então para selecionar entre naturais, recebe o tipo Nat e retorna o resultado final. Todos os booleanos tem a mesma assinatura de tipo polimórfica, independentemente do contexto em que eles se encontram. Isso faz com que a geração de código seja muito mais simples, pois não há a necessidade de propagar o tipo de retorno final para

gerar os booleanos corretos, como é o caso para a compilação para o STLC, precisa-se apenas inferir o tipo de retorno e criar uma aplicação de tipo sobre o termo que vai avaliar para um booleano. E fazendo isso, a própria linguagem irá ajustar as anotações de tipo quando reduzir todas as aplicações de tipo.

Naturais como iteradores

Agora que podemos usar polimorfismo, gerar código para `natRec` é muito mais simples. Assim como os booleanos, os naturais tem uma definição única e genérica que pode ser utilizada para iterar sobre qualquer tipo. A anotação de tipo dos naturais é `(forall C, (C->C)->C->C)`.

```
natRec(5; double; 1)

== Compila para =>

\\C. \f:C->C. \x:C. (f (f (f (f (f x))))) [Nat] double 1

\f:Nat->Nat. \x:Nat. (f (f (f (f (f x))))) double 1

(double (double (double (double (double 1)))))
```

Listing 4.5 – Funcionamento de natRec

Na Listing 4.5 temos a expressão SimpleLambda `natRec(5;double, 1)`, onde supomos que temos a disposição uma função chamada `double` que recebe um natural e retorna o seu valor multiplicado por 2. O que essa expressão SimpleLamba faz é elevar 2 a quinta potência

No código $\lambda 2$ gerado, primeiro usamos a notação de colchetes para aplicar o tipo Nat ao natural 5, que ajusta os tipos esperados pelo termo. Então com a aplicação de `double` para a variável `f`, e a aplicação de 1 para a variável `x`, temos o resultado de cinco aplicações sucessivas de `double` sobre o inicial 1, que calcula o valor de 2 na quinta potência.

O numeral zero é representado por um termo em que a função recebida não é usada, então o valor inicial (no caso do exemplo, 1) é retornado intacto.

Gerando código para Subtração e Comparadores

Com polimorfismo agora é possível gerar código para subtração e comparações. Para representar essas operações em $\lambda 2$ o processo é mais complexo, por exemplo, a subtração é implementada usando um dos números como iterador para aplicar uma função predecessor sobre o outro. E essa função predecessor, é implementada iterando sobre pares realizando uma contagem deslocada, em que o primeiro valor do par é 1 menor que o segundo. O passo dessa contagem pode ser implementado como uma função `shiftIncrease` que copia o valor da segunda posição para a primeira, e incrementa em um o valor que ficou na segunda. Então começando a contagem com `(0,0)`, ao final de n passos o valor do par será `(n-1, n)` e o predecessor de n poderá ser extraído da primeira posição do par.

Além de ser um termo bastante extenso para traduzir por inteiro, a subtração pode ser completamente descrita com construções mais simples da linguagem. Para traduzir uma soma, por exemplo, é bastante simples, basta incluir o termo fixo que representa soma no cálculo alvo, e as aplicações de seus operandos. Com subtração, no entanto, a solução mais simples foi definir uma AST que representa toda a função e chamar recursivamente a função de geração de código sobre essa AST. Isso quebra a tarefa complexa de representar subtração em várias tarefas mais simples.

Uma segunda vantagem é que essa mesma AST pode ser usada para gerar código $\lambda\omega$ e λC, pois as particularidade da sintaxe são definidas pelas funções de geração de cada cálculo. Vemos na Figura 4.10 a definição em PureScript das AST usadas para representar subtração, usando o *data type* `Term` visto na Figura 4.2.

Figura 4.10: Termos que representam subtração.

```
shiftIncTerm :: Term
shiftIncTerm = (T_func_system "p" (Pair Nat Nat)
                    (T_pair (T_snd (T_var_system "p"))
                    (T_binop  Add  (T_snd (T_var_system "p")) (T_num 1)))
                )

predTerm :: Term
predTerm  = (T_func_system "n" Nat
              (T_fst
                 (T_natRec (T_var_system "n") shiftIncTerm (T_pair (T_num 0) (T_num 0)))
              )
           )

subTerm :: Term
subTerm  = (T_func_system "n" Nat (T_func_system "m" Nat

              (T_natRec (T_var_system "m") predTerm (T_var_system "n"))

           ))
```

Tipagem de pares

Para anotar o tipo de um par de naturais, um par de booleanos e um par com um natural e um booleano, foram usados o seguintes tipos polimórficos em $\lambda 2$:

```
(forall C, (Nat  -> Nat  -> C) -> C)
(forall C, (Bool -> Bool -> C) -> C)
(forall C, (Nat  -> Bool -> C) -> C)
```

Com esses tipos retornamos a um problema familiar: assim como em STLC não tínhamos booleanos e naturais genéricos,

em $\lambda 2$ não temos como construir um tipo par ordenado genérico que representa qualquer tipo par ordenado.Os resultados são muito menos limitantes que em STLC, porém em caso de aninhamento de pares as anotações de tipos começam a ficar cada vez mais extensas e com isso os termos menos legíveis. Por exemplo essa é a anotação de tipos para um par de pares de naturais em $\lambda 2$:

```
((1,2), (3,4)) :
(forall C, ((forall C, (Nat  -> Nat  -> C) -> C)  ->
            (forall C, (Nat  -> Nat  -> C) -> C)  -> C) -> C
               )
```

Com apenas um nível de aninhamento de pares já podemos ver como crescem as anotações de tipo. Para a geração de código isso é apenas uma inconveniência. Há um padrão claro para montar esses tipos, mas ainda não temos a capacidade de abstrair esse padrão para uma definição genérica. Essa capacidade é obtida com $\lambda\omega$.

4.2.6 Cálculo Lambda Polimórfico de Alta Ordem ($\lambda\omega$)

Em $\lambda\omega$ são incluídos construtores de tipos, que é uma maneira de abstrair tipos na definição de tipos, ou seja tipos que dependem de tipos. Com isso, agora podemos criar uma definição genérica para qualquer par, que pode ser instanciada com qualquer tipo. No exemplo a seguir podemos ver a definição de `Pair` e a anotação de tipo para um par de pares de naturais:

```
Pair  = \A:*, \B:*, forall C:*, (A -> B -> C) -> C;
```

```
((1,2), (3,4)) : (Pair (Pair Nat Nat) (Pair Nat Nat))
```

Na linguagem SimpleLambda, temos dois construtores de tipos: o construtor $\times$ que dados dois tipos T_1 e T_2, constrói um tipo par ordenado $T_1 \times T_2$, e o construtor $\rightarrow$, que dados dois tipos T_1 e T_2 constrói um tipo função $T_1 \rightarrow T_2$. Caso o usuário escolha compilar para $\lambda\omega$ e optar pela compilação com definições, o código gerado será mais legível e evidenciará o papel dos construtores de tipos de SimpleLambda representados no código $\lambda\omega$.

4.2.7 Cálculo de Construções (λC)

A principal mudança para Cálculo de Construções são os tipos dependentes: temos agora a possibilidade de aplicar termos a tipos. Para a geração de código para a ferramenta, no entanto, não trás muitas modificações, pois não temos tipos dependentes em SimpleLambda. Todas as traduções para $\lambda\omega$ podem ser usadas com pequenos ajustes de sintaxe em λC, e não temos tanto ganho de legibilidade no código gerado quanto vimos na transição de $\lambda 2$ para $\lambda\omega$.

Apesar de não mudar fundamentalmente o código gerado, algumas diferenças na sintaxe do simulador resultam em um código visivelmente diferente de $\lambda\omega$ e que sugerem essas novas possibilidades ao usuário. A principal é que não há mais distinção entre aplicação de termos e tipos, então a notação de dupla barra (`\\`) para a abstração de tipos e a notação de colchetes (`[]`) para a aplicação de tipos sobre termos, não são mais usadas. Usa-se a mesma sintaxe para todas as formas de abstração e aplicação. Também é possível agora usar o operador Π (representado por barra vertical dupla, ||) para definir todas as formas de tipos funcionais, não sendo mais necessário usar a palavra `forall`.

O simulador para Cálculo de Construções, assim como os outros, oferece um bloco para definições de tipo, porém ele não é mais obrigatório. No código gerado, os tipos agora são definidos junto com os termos no bloco `let..in`, para que a diferença entre $\lambda\omega$ e λC fique mais clara.

Podemos ver todas essas diferenças sintáticas no código λC abaixo, gerado para o par (1,2) de SimpleLambda com a opção de compilação com definições ativada na ferramenta.

```
let
  Bool          = ||C:*. C -> C -> C;
  Nat           = ||C:*. (C -> C) -> C -> C;
  Pair          = \A:*. \B:*. ||C:*. (A -> B -> C) -> C;

  pair    = \A:*. \B:*. \a: A. \b: B. \C:*. \f: A->B->C. f a b;

in

(pair Nat Nat 1 2)
```

4.3 Tecnologias usadas

Para desenvolver a ferramenta foi usada uma combinação de tecnologias voltadas para o desenvolvimento na web. Todo o *back end* de compilação foi implementado em Purescript (PURESCRIPT, 2013), uma linguagem funcional fortemente tipada inspirada em Haskell (HASKELL, 1990) que compila para JavaScript. Com isso podemos programar em uma linguagem funcional moderna com todas as vantagens que isso traz, e executar o resultado nativamente no navegador. Purescript conta com uma extensa lista de bibliotecas, incluindo muitas reimplementações de bibliotecas de Haskell, incluindo a biblioteca de Parsing (PARSING, 2017) usada nesse trabalho que reimplementa a biblioteca Parsec (PARSEC, 2006).

O código escrito em PureScript é compilado para JavaScript em módulos ES, que exportam as funções definidas. Essas funções então são importadas em um arquivo JavaScript, que controla o funcionamento da página e associa cada função ao seu botão correspondente. O layout da página foi definido usando HTML e CSS.

5 CONCLUSÃO

Nesse trabalho foi desenvolvido uma aplicação web que permite ao usuário definir programas em SimpleLambda e compilar para quatro variações de Cálculo Lambda tipado. A ferramenta está disponível em <https://www.inf.ufrgs.br/~atamaral/simpleLambda/>.

Ao definir um programa em SimpleLambda, podemos alternar entre todos os alvo de compilação disponíveis, o que nos permite identificar claramente as diferenças entre os cálculos. E a partir dessas diferenças podemos apreciar como cada funcionalidade dos cálculos é usada para expressar o código original em SimpleLambda. E além disso, as opções e configurações do compilador nos permitem escolher entre um código mais legível usando as definições de termos e tipos do simulador, ou mais próximo da notação original de cada cálculo, sem as definições.

O uso da linguagem PureScript para desenvolver o compilador possibilitou que a ferramenta fosse desenvolvida em uma linguagem funcional moderna quase idêntica a Haskell, e ainda rodar online em qualquer navegador sem necessidade de instalação. A maior facilidade de acesso e distribuição é muito importante para uma ferramenta didática, e com PureScript isso foi alcançado sem aumentar a dificuldade de implementação.

A geração de código para STLC se mostrou uma tarefa bastante difícil, e poderia ter sido levado mais adiante do que foi feito nesse trabalho. Apenas os booleanos e os pares têm as suas construções adaptadas para o tipo exigido pelo contexto em que são usados. A mesma adaptação pode ser feita para os naturais, o que iria permitir compilar programas SimpleLambda com construção `natRec`. No entanto, para realmente completar a geração de código para STLC, seria necessário um estudo mais profundo sobre quais construções de SimpleLambda são possíveis de se representar em STLC, e como identificar os casos em que não é possível.

Entre as possíveis melhorias para a ferramenta, a principal seria a produção de mensagens de erro mais informativas do compilador. Atualmente os erros de tipo são anunciados com uma mensagem “Erro de Tipo”, sem fornecer qualquer informação sobre onde o erro ocorre, ou qual tipo era esperado. O mesmo acontece para erros de sintaxe, que resultam em uma mensagem de “Sintaxe Incorreta”. Como o propósito da ferramenta é didático, não é esperado que sejam desenvolvidos programas muito complexos ou muito extensos em SimpleLambda. Por isso a falta de mensagens de erro informativas não é intolerável, apesar de ser bastante inconveniente.

Um possível trabalho futuro seria estender a linguagem SimpleLambda com funcionalidades de assistente de prova. O usuário descreveria um sequente válido, e poderia aplicar regras lógicas até chegar a uma prova, e a partir disso seria gerado um programa que codifica essa prova. A base de geração de código para Cálculo Lambda tipado desenvolvida nesse trabalho poderia ser usada para montar esses termos-prova.

REFERÊNCIAS

BARENDREGT, H. Introduction to generalized type systems. **Journal of Functional Programming**, Cambridge University Press, v. 1, n. 2, p. 125–154, 1991.

CHURCH, A. A set of postulates for the foundation of logic. **Annals of Mathematics**, Annals of Mathematics, v. 33, n. 2, p. 346–366, 1932. ISSN 0003486X. Available from Internet: <http://www.jstor.org/stable/1968337>.

CHURCH, A. A formulation of the simple theory of types. **The Journal of Symbolic Logic**, Association for Symbolic Logic, v. 5, n. 2, p. 56–68, 1940. ISSN 00224812. Available from Internet: <http://www.jstor.org/stable/2266170>.

COQUAND, T.; HUET, G. The calculus of constructions. **Information and Computation**, v. 76, n. 2, p. 95–120, 1988. ISSN 0890-5401. Available from Internet: <https://www.sciencedirect.com/science/article/pii/0890540188900053>.

EXPR. **Expr: this module is a port of the Haskell Text.Parsec.Expr module.** 2017. <https://pursuit.purescript.org/packages/purescript-parsing/10.2.0/docs/Parsing.Expr>. Accessed: 2023-03-26.

GIRARD, J.-Y. Une extension de L'interpretation de gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types. In: FENSTAD, J. (Ed.). **Proceedings of the Second Scandinavian Logic Symposium**. Elsevier, 1971, (Studies in Logic and the Foundations of Mathematics, v. 63). p. 63–92. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0049237X08708437>.

GIRARD, J.-Y. **Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur**. Thesis (PhD) — Éditeur inconnu, 1972.

HASKELL. **Haskell: An advanced, purely functional programming language**. 1990. <https://www.haskell.org/>. Accessed: 2023-03-27.

HILLEBRAND, G.; KANELLAKIS, P. On the expressive power of simply typed and let-polymorphic lambda calculi. In: **Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science**. USA: IEEE Computer Society, 1996. (LICS '96), p. 253. ISBN 0818674636.

HOWARD, W. A. The formulae-as-types notion of construction. In: CURRY, H. et al. (Ed.). **To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism**. [S.l.]: Academic Press, 1980.

PARSEC. **parsec: Monadic parser combinators**. 2006. <https://hackage.haskell.org/package/parsec>. Accessed: 2023-03-26.

PARSING. **Parsing: monadic parser combinator library based on Haskell's Parsec.** 2017. <https://pursuit.purescript.org/packages/purescript-parsing/10.2.0>. Accessed: 2023-03-26.

PLOTKIN, G. Lcf considered as a programming language. **Theoretical Computer Science**, v. 5, n. 3, p. 223–255, 1977. ISSN 0304-3975. Available from Internet: <https://www.sciencedirect.com/science/article/pii/0304397577900445>.

PURESCRIPT. **PureScript: A strongly-typed functional programming language that compiles to JavaScript**. 2013. <https://www.purescript.org/>. Accessed: 2023-03-27.

REYNOLDS, J. C. Towards a theory of type structure. In: **Symposium on Programming**. [s.n.], 1974. Available from Internet: <https://www.cis.upenn.edu/~stevez/cis670/pdfs/Reynolds74.pdf>.

TOKEN. **Token: this module is a port of the Haskell Text.Parsec.Token module.** 2017. <https://pursuit.purescript.org/packages/purescript-parsing/10.2.0/docs/Parsing.Token>. Accessed: 2023-03-26.