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

EDUARDO BASSANI CHANDELIER

Simulador para o Cubo Lambda Estendido com Tipos Indutivos

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

RESUMO

A compreensão de Cálculo Lambda e Teoria dos Tipos é importante para entender com mais profundidade os fundamentos tanto de linguagens funcionais modernas como de ferramentas assistentes de prova. Ao longo dos anos uma série de propostas de cálculos tipados foram apresentadas por diferentes autores. Neste cenário, o Cubo Lambda surgiu como uma forma de sistematizar o conhecimento em torno desses cálculos. No contexto do ensino das diversas variantes do Cálculo Lambda Tipado, uma ferramenta que consiga explorar todos os oito vértices do Cubo Lambda e com uma sintaxe unificada se mostra desejável. Este trabalho descreve um simulador do Cálculo Lambda Tipado parametrizável, ou seja, que permite a experimentação com diversas funcionalidades do Cubo Lambda de forma conjunta ou independente, incluindo Polimorfismo, Construtores de Tipos, Tipos Dependentes e uma extensão dos respectivos cálculos que permite a definição de tipos indutivos como primitivas. Tipos indutivos são uma forma de definir estruturas de dados de forma recursiva muito utilizados em linguagens de programação funcionais modernas. A ferramenta pode ser encontrada no link <https://www.inf.ufrgs.br/~ebchandelier/>.

Palavras-chave: Cálculo lambda tipado. Teoria dos Tipos. Sistemas WEB.

Lambda Cube Simulator extended with Inductive Types

ABSTRACT

Understanding Lambda Calculus and Type Theory is important to deeply understand the fundamentals of both modern functional languages and proof assistant tools. Over the years different typed lambda calculus have been designed by different authors. In this scenario, the Lambda Cube emerged as a way to systematize knowledge around this subject. The Lambda Cube is a framework used to investigate the different versions of the typed Lambda Calculus. In the context of teaching the multiple variants of Typed Lambda Calculus, a tool that manages to explore all eight vertices of the Lambda Cube with a unified syntax is desirable. This work describes a parameterizable Typed Lambda Calculus simulator, that is, that allows experimentation with various functionalities of the Lambda Cube, including Polymorphism, Type Constructors, Dependent Types and an extension of the respective calculus that allows the definition of inductive types as primitives. Inductive types are a way of defining data structures recursively that are widely used in modern functional programming languages. The tool can be found at <https://www.inf.ufrgs.br/ ebchandelier/>.

Keywords: Typed Lambda Calculus. Type Theory. WEB Systems.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

$\lambda_{\rightarrow}$	Cálculo Lambda Tipado Simples (*Simply Typed Lambda Calculus*)
$\lambda 2$	Cálculo Lambda Polimórfico
$\lambda\omega$	Cálculo Lambda Com Construtores de Tipos
λ C	Cálculo de Construções
Redex	Expressão Redutível (Reducible Expression)
$\rightarrow_{\beta}$	Redução Beta
$=_{\alpha}$	Relação de Alfa Equivalência
*	Kind
$\Box$	SuperKind

SUMÁRIO

1 INTRODUÇÃO

O estudo da Teoria dos Tipos (RUSSELL, 1908) e de diversas variantes tipadas do cálculo lambda (CHURCH, 1932) se localiza na intersecção entre a computação e a lógica, sendo a base matemática para a construção de linguagens de programação funcionais modernas e também de ferramentas assistentes de provas baseadas no Isomorfismo de Curry Howard (SøRENSEN; URZYCZYN, 2006).

A compreensão de Cálculo Lambda e Teoria dos Tipos é importante para entender com mais profundidade os fundamentos tanto de linguagens funcionais modernas como de ferramentas assistentes de prova, possibilitando a implementação de compiladores e interpretadores mais robustos. A disciplina de "Tópicos especiais: Cálculo Lambda e Teoria de Tipos (INF05013)" do Instituto de Informática da UFRGS, trata sobre o tema. Os professores fazem o uso de quatro ferramentas para auxiliar no ensino, cada ferramenta voltada para um Cálculo Lambda tipado específico: O Simulador de Cálculo Lambda Tipado Simples, Simulador de Cálculo Lambda Polimórfico, Simulador de Cálculo Lambda Polimórfico de Alta Ordem e o Simulador para o Cálculo de Construções. Cada um desses simuladores possui a sua sintaxe própria e com eles é possível experimentar com apenas quatro dos oito cálculos tipados representados como vértices do Cubo Lambda (BARENDREGT, 1991). Nenhum desses quatro simuladores provê suporte para a experimentação com tipos indutivos como primitivas, embora seja possível definir tipos indutivos através de polimorfismo. Nesse contexto, para facilitar a aprendizado de Teoria dos Tipos, uma única ferramenta que consiga explorar todos os oito vértices do Cubo Lambda por meio de uma sintaxe unificada se mostra desejável.

Este trabalho contribui com a concepção e a implementação de uma única ferramenta que permite ao usuário experimentar com todos os oito vértices do Cubo Lambda e também com uma extensão com suporte nativo a tipos indutivos. O usuário poderá, de forma prática, avaliar o poder de expressão dos cálculos e, dessa forma, aprofundar seus conhecimentos sobre polimorfismo, construtores de tipos, tipos dependentes e tipos indutivos, e a relação desses conceitos com linguagens de programação e também com assistentes de prova.

O simulador pode ser acessado através do endereço <https://www.inf.ufrgs.br/~ebchandelier/>. O seu código fonte está disponível em <https://bitbucket.org/ebchandelier/lambda-calculus-simulator/src/master/>.

Este trabalho está organizado da seguinte forma: os capítulos 2, 3 e 4 apresentam

uma visão geral sobre conceitos necessários sobre Cálculo Lambda e Teoria dos Tipos para o uso da ferramenta. Em particular, o capítulo 2 dá uma breve introdução sobre o Cálculo Lambda puro, o Capítulo 3 apresenta uma visão geral sobre Cálculos Lambda tipados e sobre o Cubo Lambda, e o Capítulo 4 discute tipos indutivos. Por fim, a ferramenta é apresentada no Capítulo 5. O trabalho encerra com Conclusões e discussão sobre trabalho futuros.

2 CÁLCULO LAMBDA SEM TIPOS

O Cálculo Lambda foi criado por Alonzo Church no ano de 1932 (CHURCH, 1932). Alan Turing posteriormente propôs o modelo de computação que veio a ser conhecido como Máquina de Turing, sendo ambos modelos provadamente equivalentes quanto a seu poder de expressão (TURING, 1936).

O Cálculo Lambda é um sistema formal para expressar computação. Linguagens de programação usuais expressam computação usando funções, atribuição, condicionais, laços, recursão, classes, objetos, booleanos, inteiros, e outros. Já o Cálculo Lambda expressa computação usando somente funções e aplicação de funções a argumentos, podendo ser considerado um dos menores modelos de computação.

A sintaxe abstrata do Cálculo Lambda é dada pela gramática abstrata abaixo (2.1).

$$M ::= x \mid \lambda x.M \mid MN \tag{2.1}$$

Um termo lambda pode ser uma variável x, uma abstração lambda $\lambda x.M$ que define função e liga todas as ocorrências livres de x em M e a aplicação entre termos $M\ N$. Uma variável x qualquer é dita ligada quando está em M no termo $\lambda x.M$ caso contrário, é dita livre.

Quando uma aplicação lambda possui uma abstração lambda a esquerda, ou seja, um termo na forma $(\lambda x.M)N$, esse termo é uma expressão redutível (*reducible expression*, ou, de forma curta, *redex*) e pode, então, sofrer uma redução beta. A regra de redução beta abaixo (β) transforma o *redex* do lado esquerdo de $\rightarrow_\beta$ no termo $[Q/x]P$, ou seja, no termo P com todas as ocorrências livres de x são substituídas pelo termo Q. Quando um termo não possui mais nenhum *redex*, diz-se que ele está em sua forma normal.

$$(\lambda x.P)\ Q \rightarrow_\beta [Q/x]P \tag{β}$$

Um exemplo de redução beta pode ser visto em $(\lambda x.x\ x)(\lambda c.c) \rightarrow_\beta (\lambda c.c)(\lambda c.c)$, onde todas as variáveis x no termo $x\ x$ são substituídas por $\lambda c.c$ resultando no termo $(\lambda c.c)(\lambda c.c)$, este último podendo dar mais um passo de redução, com a substituição da variável c do corpo da função à esquerda da aplicação pelo termo à direita da aplicação, ou seja, $(\lambda c.c)(\lambda c.c) \rightarrow_\beta (\lambda c.c)$.

A forma de avaliar um termo lambda, conhecido como *full beta reduction*, avalia de forma sequencial ou paralela todos os *redexes* de um termo, independente de onde eles ocorram, até que uma forma normal seja produzida, caso ela exista. Dito isto, exis-

tem duas estratégias principais de avaliação que ditam uma ordem específica em que os *redexes* devem ser executados: a estratégia normal, que avalia o *redex* mais externo e à esquerda do termo, e a estratégia aplicativa, que avalia o *redex* mais interno e mais à esquerda do termo.

Dois termos, M e N, quando só diferem no nome de suas variáveis ligadas, são chamados de alfa-equivalentes, e escrevemos $M =_\alpha N$ para representar esse fato. Por exemplo $(\lambda a.a) =_\alpha (\lambda b.b)$.

O Cálculo Lambda respeita as seguintes propriedades:

- Confluência (Church-Rosser): Se $M \to N_1$ e $M \to N_2$ então existe um P onde $N_1 \to P$ e $N_2 \to P$. Ou seja, para qualquer termo, se ele beta reduz para dois termos distintos, esses termos distintos eventualmente vão beta reduzir para um termo em comum.
- Normalização: Quando utilizamos a estratégia normal de redução, é garantido que o termo sempre chega em sua forma normal, quando ela existe.

Um exemplo de codificação muito conhecido são os numerais de Church, que define um natural *n* como o termo $\lambda f.\lambda x.f^n\ x$, onde a notação f^n representa a função f aplicada *n* vezes em x. Por exemplo, 0 = $\lambda f.\lambda x.x$, 1 = $\lambda f.\lambda x f.x$, dois = $\lambda f.\lambda x f(fx)$, etc.

No próximo capítulo veremos uma série de cálculos lambda tipados. Todos esses cálculos apresentam uma estrutura básica semelhante à do cálculo lambda puro visto nesse capítulo, a saber possuem variáveis, funções e aplicação de um termo a outro. Os cálculos tipados, porém, são todos fortemente normalizáveis, o que significa que não são Turing Completos.

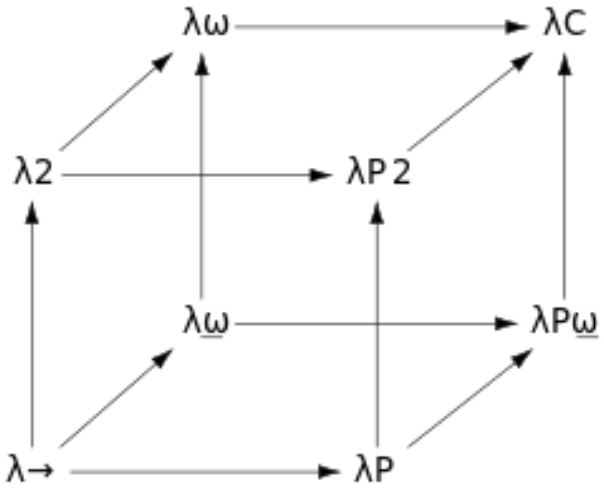
3 CÁLCULOS LAMBDA TIPADOS E O CUBO LAMBDA

Após a invenção do Cálculo Lambda por Church, diversas versões tipadas foram desenvolvidas por inúmeros pesquisadores ao longo dos anos. Dentre essas versões podemos mencionar o cálculo lambda tipado simples, proposto pelo próprio Church (CHURCH, 1940); cálculo lambda polimórfico desenvolvido de forma independente por Reynolds (REYNOLDS, 1974), um cientista da computação, e também por Girard (GIRARD, 1972), um lógico matemático, que o chamou de *System F*; o cálculo lambda polimórfico com construtores de tipos; o cálculo lambda tipado com tipos dependentes (COQUAND; HUET, 1988) e outras variações possíveis.

Com o intuito de sistematizar o conhecimento adquirido ao longo dos anos pelas diversas propostas de cálculos lambda tipados, Barendregt (BARENDREGT, 1991) propôs o que veio a ser conhecido como o Cubo Lambda, representado na Figura 3.1.

Cada vértice do Cubo representa um Cálculo Lambda tipado. No vértice inferior à esquerda temos o Cálculo Lambda Tipado Simples ($\lambda_{\rightarrow}$), o mais simples de todos os oito cálculos do cubo. E no vértice superior à direita temos o Cálculo de Construções (λC) que incorpora todos os recursos dos demais cálculos.

Figura 3.1: Cubo Lambda



Fonte: <https://en.wikipedia.org/wiki/Lambda_cube>

Diversas características, tais como polimorfismo, construtores de tipos e tipos indutivos, tiveram origem em estudos das extensões tipadas do Cálculo Lambda. Tipos dependentes, uma extensão na qual tipos podem depender de termos, ainda é pouco explorada em linguagens de programação, podendo ser a próxima contribuição da teoria dos tipos para essa área. Teoria dos Tipos é uma área que faz intersecção entre com-

putação e lógica matemática. A dualidade entre lógica e programação, que interpreta tipos como proposições e programas como provas, ficou conhecida como Isomorfismo de Curry-Howard (SøRENSEN; URZYCZYN, 2006). E, nesse contexto, tipos dependentes são essenciais em assistentes de provas baseados no Isomorfismo de Curry-Howard, tais como Coq (BERTOT; CASTÉRAN, 2013) e Lean (MOURA et al., 2015).

Todos os cálculos lambda tipados que serão vistos a seguir apresentam as seguintes propriedades:

- Normalização: A avaliação de todo termo bem tipado sempre termina com a sua forma normal, independente da estratégia de avaliação utilizada.
- Preservação de Tipo: Para qualquer termo bem tipado, o tipo é preservado quando são realizadas beta reduções.

Nas demais seções serão apresentados, em ordem de complexidade, alguns vértices importantes do Cubo.

Embora a ferramenta propriamente dita será apresentada somente no Capítulo 5, alguns exemplos de termos dos cálculos das próximas seções serão mostrados já utilizando a interface do simulador. No Capítulo 4 será apresentada a extensão com tipos indutivos.

3.1 Cálculo Lambda Tipado Simples - $\lambda_{\rightarrow}$

O Cálculo Lambda Tipado Simples, o qual será denotado deste ponto em diante por $\lambda_{\rightarrow}$, foi criado por Alonzo Church em 1940 (CHURCH, 1940) como uma variante do cálculo lambda sem tipos. O modelo original foi estendido com assinatura de tipos, agora permitindo ter mais controle sobre a aplicação dos termos.

A sintaxe e a semântica do Cálculo Lambda foram modificadas para suportar a tipagem de variáveis ligadas em abstrações lambda. A sintaxe do $\lambda_{\rightarrow}$ é dada pela gramática abstrata (3.1) e (3.2) abaixo:

$$M ::= x \mid MN \mid \lambda x : \boldsymbol{\tau} \,.\, M \tag{3.1}$$

$$\boldsymbol{\tau} ::= \boldsymbol{X} \mid \boldsymbol{\tau_1} \rightarrow \boldsymbol{\tau_2} \tag{3.2}$$

Foi adicionada uma gramática específica para tipos representados pela letra τ

(3.2), onde X representa um tipo atômico e $\tau_1 \rightarrow \tau_2$ é um tipo função com τ_1 o tipo de entrada e τ_2 o tipo de saída. E aos termos foi adicionado a assinatura de tipos na abstração lambda $\lambda x : \tau.M$ (3.1).

Abaixo seguem as regras de tipo para as construções do $\lambda_{\rightarrow}$. Nas regras, Γ é chamada de contexto e mapeia variáveis para seus tipos. A notação $\Gamma \vdash M : \tau$, chamada de julgamento de tipo, é lida como: *o termo M tem o tipo τ sob o contexto de tipo Γ*.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{(VAR)}$$

$$\frac{\Gamma \vdash M : \tau \rightarrow \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M\ N : \tau'} \qquad \text{(APP)}$$

$$\frac{\Gamma,\ x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x : \tau.M : \tau \rightarrow \tau'} \qquad \text{(ABS)}$$

Como a expressividade do $\lambda_{\rightarrow}$ é muito limitada, normalmente ele é estudado ou ensinado com outras extensões. Existem diversas extensões que podem ser feitas adicionando certas primitivas ao cálculo, como naturais, booleanos, produto, união ou até mesmo operador de ponto fixo, que adiciona à linguagem a possibilidade de representar recursão geral. Umas das mais famosas variações do $\lambda_{\rightarrow}$, que aumentam expressividade da linguagem para diferentes propósitos, são o Gödel's *System T* (GöDEL, 1958) e o *PCF* (PLOTKIN, 1977), sendo esta última a base para muitas linguagens de programação.

Com a extensão de assinatura de tipos, um exemplo de termo que pode ser codificado em Cálculo Lambda, mas que não pode ser representado em nenhum outro Cálculo Tipado é o Combinador Y, que serve para construir funções recursivas, definido como $Y = \lambda f.(\lambda x.(f(x\ x))\lambda x.(f(x\ x)))$. Com isso, o $\lambda_{\rightarrow}$ não possui a propriedade de universalidade, ou seja, não é mais equivalente em poder de expressão à Máquina de Turing ou ao Cálculo Lambda.

Pelo Isomorfismo de Curry Howard, os tipos do $\lambda_{\rightarrow}$ correspondem a fórmulas da lógica proposicional minimal, e um termo do $\lambda_{\rightarrow}$ bem tipado corresponde a uma prova da fórmula correspondente ao tipo do termo. A Figura 3.2 apresenta um exemplo de prova que pode ser feita usando as regras de tipagem do $\lambda_{\rightarrow}$.

Essa mesma derivação em forma de árvore da Figura 3.2 pode ser construída no simulador conforme mostra a Figura 3.3. Notar que a proposição, ou tipo, nesse exemplo é o $(A \rightarrow B) \rightarrow A \rightarrow B$ (linha 2), e a prova desta proposição, ou o termo desse tipo pode

Figura 3.2: Prova em forma de Árvore - Prova da fórmula $(A \to B) \to A \to B$ em $\lambda \to$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(y) = A \to B}{\Gamma \vdash y : A \to B}\,\text{VAR} \quad \dfrac{\Gamma(z) = A}{\Gamma \vdash z : A}\,\text{VAR}}{y : A \to B, z : A \vdash y\ z : B}\,\text{APP}}{y : A \to B \vdash \lambda z : A.\ y\ z : A \to B}\,\text{ABS}}{\vdash \lambda y : A \to B.\ \lambda z : A.\ y\ z : (A \to B) \to A \to B}\,\text{ABS}$$

ser visto na linha 8. Detalhes do simulador serão discutidos no Capítulo 5.

Figura 3.3: Exemplo no simulador - Construção de uma prova para a proposição $(A \to B) \to A \to B$ em $\lambda \to$

```
1  typedef
2    Proposition = (A -> B) -> A -> B;
3  end
4  var
5    A:*; B:*;
6  end
7  let
8    proof = \ab:(A -> B). \a:A. ab a;
9  in
10   (\p:Proposition. p) proof
```

Fonte: Simulador Cálculo Lambda

O $\lambda_{\to}$ é versão mais simples das versões do cálculo cambda com tipos, e também é a menos expressiva, com pouca variedade de programas que podem ser codificados com ele. Além disso, a lógica isomórfica a ele é a lógica proposicional minimal, que possui apenas a implicação como conetivo. Uma das possíveis modificações que podem ser feitas à linguagem para aumentar a expressividade é a adição de polimorfismo.

3.2 Cálculo Lambda Polimórfico - $\lambda 2$

O Cálculo Lambda Polimórfico ($\lambda 2$) foi criado em 1972 pelo matemático Jean Yves Girard (GIRARD, 1972) por um ponto de vista da lógica com o nome de *System F* e por John Reynolds em 1974 (REYNOLDS, 1974) de um ponto de vista da computação com o nome de Cálculo Lambda Polimórfico. Esse cálculo adiciona a possibilidade de criar termos que recebem tipos como argumentos, ou seja, passar tipos como argumentos de termos estendendo dessa forma o $\lambda_{\to}$ que só admite termos que recebem termos como argumentos.

A gramática abstrata de $\lambda 2$ pode ser vista em 3.3 e 3.4, com as diferenças entre o

$\lambda_{\rightarrow}$ e $\lambda 2$ destacadas em negrito.

$$M \ ::= \ x \ \mid \ MN \ \mid \ \lambda x : \tau . M \ \mid \ \boldsymbol{\Lambda X . M} \ \mid \ \boldsymbol{M \ \langle \tau \rangle} \tag{3.3}$$

$$\tau \ ::= \ X \ \mid \ \tau_1 \rightarrow \tau_2 \ \mid \ \boldsymbol{\forall X . \tau} \tag{3.4}$$

São adicionados aos termos a abstração lambda que recebe um tipo e retorna um termo $\Lambda X : M$ e a aplicação entre termos e tipos $M \ \langle \tau \rangle$. Em muitas apresentações do $\lambda 2$, as abstrações lambda recebem notações do símbolo lambda diferentes para distinguir os tipos de entrada e saída, sendo o símbolo lambda minusculo (λ) usado para representa abstração de termos em termos e o símbolo lambda maiúsculo (Λ) usado para representar a abstração de tipos em termos. Aos tipos é adicionado o tipo $\forall X : \tau$ que é o tipo de um termo $\Lambda X : M$, supondo que M seja do tipo τ. Na linguagem de tipos, X passa a ser uma variável de tipo e um tipo com variáveis de tipo quantificadas universalmente é um tipo polimórfico.

Linguagens como OCaml, Haskel e outras possuem uma versão restrita de polimorfismo pela qual a quantificação de variáveis de tipo só pode ocorrer na posição mais externa de um tipo. Isso se deve ao fato de que o problema da reconstrução de tipos do $\lambda 2$ é indecidível (WELLS, 1999).

No $\lambda_{\rightarrow}$ existe uma função identidade para cada tipo, por exemplo, $\lambda x : X.x$, $\lambda x : Y.x$, $\lambda x : Z.x$, etc. Com o $\lambda 2$ é possível termos uma única função identidade polimórfica que opera com vários tipos $\Lambda X.\lambda x : X.x$ e essa função identidade é do tipo $\forall X.X \rightarrow X$.

A Figura 3.4 mostra a codificação de booleanos e operações sobre booleanos no simulador usando polimorfismo. A uso de polimorfismo pode ser vista na linha 5, 6 e 7, onde são usadas abstrações lambda que recebem tipos como argumentos.

É possível implementar fórmulas com operadores lógicos como $\wedge$ ou $\vee$ usando polimorfismo ou, de forma equivalente, tipos pares ordenados e união disjunta. A Figura 3.5 mostra uma implementação no simulador da prova da proposição $(A \wedge B) \rightarrow (A \vee B)$.

No $\lambda 2$ é possível representar várias estruturas que não eram possíveis no $\lambda_{\rightarrow}$, entretanto algumas estruturas não podem ser representadas de forma genérica. Por exemplo, para representar o tipo $A \wedge B$ é necessário codificar os construtores e destrutores específicos para o tipo $AndAB$. Para facilitar a programação no nível de tipos, fica evidente a necessidade da possibilidade da construção de funções ao nível de tipos, ou seja, constru-

Figura 3.4: Exemplo no simulador - Operações sobre Booleanos em $\lambda 2$

```
typedef
  Bool = forall C:*. C -> C -> C;
end
let
  true = \C:*. \a:C. \b:C. a;
  false = \C:*. \a:C. \b:C. b;
  if = \C:*. \cond:Bool. \caseT:C. \caseF:C. cond C caseT caseF;

  and = \a:Bool. \b:Bool. if Bool a (if Bool b true false) false;
  or = \a:Bool. \b:Bool. if Bool a true (if Bool b true false);
  not = \b:Bool. if Bool b false true;
  then = \a:Bool. \b:Bool. if Bool a
in
  and (or true false) (not false)
```

Fonte: Simulador Cálculo Lambda Polimórfico

Figura 3.5: Exemplo no simulador - $(A \wedge B) \rightarrow (A \vee B)$

```
typedef
  AndAB = forall C:*. (A -> B -> C) -> C;
  OrAB  = forall C:*. (A -> C) -> (B -> C) -> C;
end
var
  A:*;B:*;
end
let
  productAB =  \a:A. \b:B. (\C:*. \f:(A -> B -> C). f a b);
  fstAB = \p:AndAB. p A (\a:A. \b:B. a);
  RightAB = \a:A. \C:*. \ac:(A -> C). \bc:(B -> C). ac a;

  Proposition = AndAB -> OrAB;
  proof = \andAB:AndAB. RightAB (fstAB andAB)
in

   (\p:Proposition. p)  proof
```

Fonte: Simulador Cálculo Lambda

tores de tipos que, ao receberem tipos, produzem tipos completos.

3.3 Cálculo Lambda Polimórfico de Alta Ordem - $\lambda\omega$

Ao adicionar construtores de tipos ao $\lambda 2$, chegamos ao Cálculo Lambda Polimórfico de Alta Ordem, ou $\lambda\omega$, também conhecido como *System* $F\omega$.

No $\lambda\omega$ é adicionada a funcionalidade de criar tipos que recebem tipos como parâmetros. Como agora tipos podem ser funções que recebem tipos como argumentos é adicionado a noção de *kind*, ou *Tipo dos Tipos*, representado pela letra κ. O *kind* de um tipo completo é $*$, e o *kind* de uma função que vai construir um tipo usando outro tipo é $* \rightarrow *$. A gramática abstrata de $\lambda\omega$ pode ser vista em 3.5, 3.6 e 3.7, com as diferenças entre o $\lambda 2$ e $\lambda\omega$ destacadas em negrito.

$$M \ ::= \ x \ \mid \ MN \ \mid \ \lambda x : \tau . M \ \mid \ \Lambda X : \boldsymbol{\kappa}. \ M \ \mid \ M \ \langle \tau \rangle \tag{3.5}$$

$$\tau \ ::= \ X \ \mid \ \tau_1 \rightarrow \tau_2 \ \mid \ \forall X : \boldsymbol{\kappa}. \ \tau \ \mid \ \boldsymbol{\lambda x : \kappa . \ \tau} \ \mid \ \boldsymbol{\tau_1 \ \tau_2} \tag{3.6}$$

$$\boldsymbol{\kappa \ ::= \ * \ \mid \ \kappa_1 \rightarrow \kappa_2} \tag{3.7}$$

Aos termos é adicionada a assinatura de $Kinds$, representado pela letra κ na abstração de tipos $\Lambda X \ : \ \kappa . M$, e aos tipos é adicionada a assinatura de $Kinds$ ao tipo $\forall X : \kappa . \tau$, e dois novos tipos: abstração de tipos $\lambda X : \kappa . \tau$ e aplicação entre tipos $\tau_1 \ \tau_2$.

Com a adição de uma abstração lambda e aplicação entre tipos no nível de tipo, agora é necessário a operação de beta redução, ou substituição, ao nível de tipos.

A Figura 3.6 mostra a prova no simulador da proposição $(A \wedge B) \rightarrow C \rightarrow ((B \vee C) \wedge \bot)$ usando o $\lambda\omega$. Nas linhas 3, 4 e 8 é possível ver a abstração lambda ao nível de tipos, possibilitando a construção de tipos mais genéricos. Tipos no $\lambda 2$ como *AndAB* agora podem ser mais genéricos como *And*, e instanciados com os tipos desejáveis, como em *And X Y*.

O $\lambda\omega$ é usado como ponto de partida para diversas linguagens de programação modernas, sendo o suficiente para representar tipos polimórficos e construtores de tipos tais como *Maybe*, *List* e *Either* presentes em linguagens de programação como OCaml e Haskell. Para usar o $\lambda\omega$ como uma linguagem de programação é necessário apenas extensões de tipos primitivos de máquina para melhorar o desempenho, como int, bool, float, double, char, e recursão genérica para aumentar o poder de expressividade. Já para um assistente de provas, ainda falta um recurso muito importante, a possibilidade de construir provas para a lógica de predicados, o que é obtido com a inclusão de tipos dependentes, ou seja, tipos que esperam termos como argumentos.

3.4 Cálculo de Construções - λC

Criado por Thierry Coquand em 1985 (COQUAND; HUET, 1988), o Cálculo de Construções, ou λC, adiciona tipos dependentes ao $\lambda\omega$. Nele é adicionado a possibilidade de construir tipos usando termos como parâmetros. A gramática abstrata de λC pode ser vista em 3.8, 3.9 e 3.10, com a diferença entre o $\lambda\omega$ e λC destacados em negrito.

Figura 3.6: Exemplo no simulador em $\lambda\omega$ - Prova da Proposição $(A \wedge B) \rightarrow C \rightarrow ((B \vee C) \wedge \bot)$

```
typedef
    Bool = forall X:*. X -> X -> X;
    Product = \A:*. \B:*. (forall C:*. (A -> B -> C) -> C);
    Either = \A:*. \B:*. forall C:*. (A -> C) -> (B -> C) -> C;

    True = forall X:*. X -> X;
    False = forall X:*. X;
    Not = \C:*. C -> False;

end
var
  A:*; B:*; C:*;
end
let
    botElim = \x:*. \c:False. x c;

    pair = \A:*. \B:*. \a:A. \b:B. \C:*.
      \abc:(A -> B -> C). abc a b;

    fst = \A:*. \B:*. \p:(Product A B). p A (\a:A. \b:B. a);
    snd = \A:*. \B:*. \p:(Product A B). p B (\a:A. \b:B. b);

    Right = \A: *. \B:*. \a:A. \C:*.
      \ac:(A -> C). \bc:(B -> C). ac a;
    Left =  \A: *. \B:*. \b:B. \C:*.
      \ac:(A -> C). \bc:(B -> C). bc b;

    case = \A:*. \B:*. \C:*. \t:(Either A B).
      \r:(A -> C). \l:(B -> C). t C r l;

    Proposition = (Product A B) -> C ->
      (Either (Product B C) False)
in
  (\x:Proposition. x)
  \pAB:(Product A B). \c:C.
    Right (Product B C) False (pair B C (snd A B pAB) c)
```

Fonte: Simulador Cálculo Lambda

$$M \ ::= \ x \ \mid \ MN \ \mid \ \lambda x : \tau . M \ \mid \ \Lambda X : \kappa .\ M \ \mid \ M \ \langle \tau \rangle \tag{3.8}$$

$$\tau \ ::= \ X \ \mid \ \tau_1 \rightarrow \tau_2 \ \mid \ \forall X : \kappa \ \tau_1 \ \mid \ \lambda x : \kappa .\ \tau \ \mid \ \tau_1 \ \tau_2 \ \mid \ \boldsymbol{\Lambda x : \tau_1 .\ \tau_2} \ \mid \ \boldsymbol{\tau \ \langle M \rangle} \tag{3.9}$$

$$\kappa \ ::= \ * \ \mid \ \kappa_1 \rightarrow \kappa_2 \tag{3.10}$$

À gramática são adicionados ao nível de tipo a abstração que recebe um termo como parâmetro $\Lambda x : \tau_1 .\ \tau_2$ e a aplicação entre tipos e termos $\tau \ \langle M \rangle$.

Agora é possível expressar nos tipos algumas propriedades dos programas a serem verificadas em tempo de compilação no momento da verificação de tipos, por exem-

plo, funções de concatenar listas que garantem o tamanho da lista de retorno como em *concat(a:List<n>, b:List<m>): List<n+m>*, onde o tipo *List<n>* significa uma lista de tamanho *n*.

Do ponto de vista da lógica, a introdução de tipos dependentes possibilita a prova de fórmulas da lógica de predicados, essencial para assistentes de prova modernos como Coq (BERTOT; CASTÉRAN, 2013) e LEAN (MOURA et al., 2015). Um exemplo de prova que pode ser feita no λC é a da fórmula $\forall x : S.\ \forall y : S. Q\ x\ y \rightarrow \forall u : S.\ Q\ u\ u$. Nessa fórmula, o símbolo predicativo Q é, na verdade, um construtor de tipos que depende de dois termos do tipo S para construir uma fórmula (um tipo). A prova desta fórmula feita no simulador está na Figura 3.7.

Figura 3.7: Exemplo no simulador - Prova de $\forall x : S.\ \forall y : S. Q\ x\ y \rightarrow \forall u : S.\ Q\ u\ u$

```
1  var
2    S: ⋆;
3    Q: S →S→⋆;
4  end
5  let
6    A = ∀x:S. ∀y:S. Q x y;
7    B = ∀u:S. Q u u;
8    proof = λz:(∀x:S. ∀y:S. Q x y). λu:S. (z u ) u
9  in
10   (λp:(A →B). p ) proof
```

Fonte: Simulador Cálculo Lambda

Apesar de ainda não ser usado em linguagens de programação populares, é possível observar uma tendência em utilizar tipos dependentes, ou versões limitadas de tipos dependentes.

Por exemplo, em *TypeScript*, uma linguagem de programação que se baseia em *JavaScript* adicionando tipos à linguagem, possui o tipo *Omit*, que tem a função de criar um tipo removendo alguma propriedade desse tipo como em 3.11, pode ser visto como uma versão tímida de tipos dependentes.

$$\{a : string\} = Omit < \{a : string, b : number\},\ 'b' > \tag{3.11}$$

Como o segundo parâmetro é uma *string*, que é um termo em sua forma normal, pode-se dizer que é possível criar tipos usando termos em *Typescript*, mas não se pode dizer que a linguagem possui tipos dependentes, pois esse parâmetro é muito limitado, podendo ser *string*, *booleano* ou *inteiro*.

O Cálculo de Construções, que usa todas as extensões, fica com uma gramática

abstrata mostrada em 3.8, 3.9 e 3.10, onde cada um dos níveis possui suas regras de semântica e tipagem, o que faz com que a linguagem seja muito complexa. Se torna desejável uma simplificação da sintaxe que englobe todas as extensões vistas até aqui.

3.5 Cubo Lambda

Criado por Henk Barendregt em 1991 (BARENDREGT, 1991), o Cubo Lambda 3.1 sistematizou todas as variações tipadas do cálculo lambda. Cada vértice do cubo representa um cálculo, e as arestas representam a adição de um dos recursos, como polimorfismo ($\uparrow$), construtores de tipos ($\nearrow$) e tipos dependentes ($\rightarrow$). Dessa forma, o vértice correspondente ao λC contempla todas as funcionalidades dos demais cálculos.

Barendregt sistematizou então todas as variantes do Cálculo Lambda em uma sintaxe unificada. Todo o cálculo de construções pode ser construído como uma única gramática abstrata (3.12). Agora não existe mais a noção de níveis (termos, tipos e *kinds*), só há uma única categoria sintática que, por uma questão de conveniência, vamos chamar de termos.

$$M \ ::= \ * \ | \ \Box \ | \ x \ | \ MN \ | \ \lambda x : M.N \ | \ \Pi x : M.N \tag{3.12}$$

Um termo agora é definido como: $*$ que significa o tipo de um tipo; $\Box$ que significa o tipo de um $Kind$; x para representar as variáveis; a aplicação entre termos $M \ N$; a abstração lambda $\lambda x \ : \ M.N$; e o tipo função $\Pi x \ : \ M.N$, ou ***Pi Type*** em inglês, que captura os quatro tipos funções vistos anteriormente. Observar que a abstração lambda e o *Pi Type* presentes na gramática abstrata em 3.12 capturam as quatro formas de abstração presentes no cálculo de construções (λD) e os quatro tipos funcionais correspondentes (ver Figura 3.8).

Figura 3.8: Tabela de Tipos funcionais para cada variação da abstração lambda

Abstração	Tipo Função	Condição
$\lambda x : \tau_1 . e_2$	$\Pi x : \tau_1 . \tau_2$	Assumindo $e_2 : \tau_2$
$\lambda X : \kappa_1 . e_2$	$\Pi X : \kappa_1 . \tau_2$	Assumindo $e_2 : \tau_2$
$\lambda X : \kappa_1 . \tau_2$	$\Pi X : \kappa_1 . \kappa_2$	Assumindo $\tau_2 : \kappa_2$
$\lambda x : \tau_1 . \tau_2$	$\Pi x : \tau_1 . \kappa_2$	Assumindo $\tau_2 : \kappa_2$

Ao sistema de tipos é adicionado a noção de *superkind*, ou $\Box$, que significa o tipo de um *kind* e pode ser visto na regra abaixo:

$$\frac{}{\Gamma \vdash * : \Box} \quad \text{(SORT)}$$

O $\Box$ é utilizado então no sistema de tipos para garantir a tipagem de termos em diferentes cálculos. A regra em que ele é utilizado é a do termo $\Pi x : A.B$ abaixo.

$$\frac{\Gamma \vdash A : s_1 \;\; \Gamma,\; x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2} \quad \text{(FORM)}$$

Na regra acima (s_1, s_2) são as possíveis combinações que permitem explorar um cálculo lambda tipado específico. (*,*) significa a aplicação de termos em termos, ($\Box$, *) a aplicação de tipos em termos, ($\Box$, $\Box$) a aplicação de tipos em tipos e (*, $\Box$) a aplicação de termos em tipos.

No capítulo seguinte veremos como estender o λC com tipos indutivos como primitivas na linguagem. Tipos indutivos são introduzidos como primitivas por uma questão de conveniência do ponto de vista de programação e por necessidade do ponto de vista do uso do simulador como uma ferramenta assistente de prova.

4 TIPOS INDUTIVOS

Tipos indutivos são tipos construídos a partir de um conjunto finito de construtores declarados, onde cada construtor pode receber diversos parâmetros de qualquer tipo, inclusive do próprio tipo sendo definido. Um elemento de um tipo indutivo é construído usando uma combinação de seus construtores. A introdução de tipos indutivos como primitivas nos possibilita representar tipos de dados mais complexos de forma mais legível, realizar computação com seus construtores e destrutores e, diferente do λC, possibilita a realização de provas por indução ao se fazer uso do simulador como um assistente de provas.

Por exemplo, os naturais em definiçôes indutivas seriam representados da seguinte forma (ver 4.1), onde Nat é o nome do tipo indutivo, O é o construtor do natural zero e S é o construtor do sucessor de um natural que recebe outro natural como parâmetro, alguns exemplos de naturais definidos nessa codificação seriam 0 = O, 1 = SO, 2 = S(SO), etc.

$$\mathrm{Nat} = \ O : \mathrm{Nat} \ \mid \ S : \mathrm{Nat} \to \mathrm{Nat} \tag{4.1}$$

Para cada novo tipo indutivo são adicionados à linguagem seus construtores e destrutores, onde os destrutores são chamados de recursores e são um mecanismo utilizado para operar sobre o tipo sendo definido. Cada um desses construtores e destrutores possui regras específicas de tipagem e semântica. Para o caso dos naturais, são adicionados à linguagem os construtores O e S e o destrutor $\mathrm{Rec}_{\mathrm{Nat}}$, e as seguintes regras de tipagem:

$$\frac{}{\Gamma \vdash O : \ \mathrm{Nat}} \tag{Nat-O}$$

$$\frac{}{\Gamma \vdash S : \mathrm{Nat} \to \mathrm{Nat}} \tag{Nat-S}$$

$$\frac{\Gamma \vdash M_1 : \mathrm{Nat} \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau \to \tau}{\Gamma \vdash \mathrm{Rec}_{\mathrm{Nat}}\ \tau \ \ M_1 \ \ M_2 \ \ M_3 : \ \tau} \tag{Rec-Nat}$$

Outra regra de tipagem que pode ser utilizada no termo $\mathrm{Rec}_{\mathrm{Nat}}$ é a que possibilita a prova por indução. A regra pode ser vista abaixo:

$$\frac{\Gamma \vdash P : \mathrm{Nat} \to * \qquad \Gamma \vdash M_1 : \mathrm{Nat} \qquad \Gamma \vdash M_2 : P\ O \qquad \Gamma \vdash M_3 : \Pi x : \mathrm{Nat}.Px \to P\ (S\ x)}{\Gamma \vdash \mathrm{Rec}_{\mathrm{Nat}}\ \tau \ \ M_1 \ \ M_2 \ \ M_3 : \ \Pi x : \mathrm{Nat}.P\ x} \tag{Rec-Nat-IND}$$

Onde para algum predicado P é necessário construir um termo para o tipo base $P\ O$ e para o passo de indução $\Pi x : \mathrm{Nat}.P\ x \to P\ (S\ x)$, gerando assim a um termo do tipo $\Pi x : \mathrm{Nat}.P\ x$. A conclusão da regra acima afirma que $\mathrm{Rec}_{\mathrm{Nat}}\ \tau\ M_1\ M_2\ M_3$ é um termo que representa uma prova de que $\Pi x : Nat.P\ x$, ou seja, de que todo natural x possui a propriedade P desde que todas as premissas sejam verdadeiras, a saber (da esquerda para a direita): P é um predicado sobre os naturais, M_1 é um natural, M_2 é uma prova de que O possui a propriedade P (base da indução) e M_3 é uma prova do passo indutivo, ou seja uma prova de que para todo natural x, se x possui a propriedade P então o sucessor $S\ x$ também possui.

A semântica operacional da linguagem precisa ser atualizada com quatro regras de avaliação abaixo.

$$\frac{M \to M'}{S\ M \to_\beta S\ M'} \qquad \text{(Nat-S)}$$

$$\frac{M \to M'}{\mathrm{Rec}_{\mathrm{Nat}}\ \tau\ M\ M_2\ M_3 \to_\beta \mathrm{Rec}_{\mathrm{Nat}}\ \tau\ M'\ M_2\ M_3} \qquad \text{(Rec-Nat-STEP)}$$

$$\frac{}{\mathrm{Rec}_{\mathrm{Nat}}\ \tau\ O\ M_2\ M_3 \to_\beta M_2} \qquad \text{(Rec-Nat-O)}$$

$$\frac{}{\mathrm{Rec}_{\mathrm{Nat}}\ \tau\ (S\ M_1)\ M_2\ M_3 \to_\beta M_3(\mathrm{Rec}_{\mathrm{Nat}}\ \tau\ M_1\ M_2\ M_3)} \qquad \text{(Rec-Nat-S)}$$

O uso de tipos indutivos, ou tipos recursivos, em linguagens de programação funcionais é praticamente um pré-requisito quando se está aprendendo a programar nesse tipo de linguagem. Em assistentes de provas também são muito utilizados para construir provas por indução.

Como o objetivo de tipos indutivos em linguagens de programação e assistentes de provas são diferentes, a forma de se usar também é diferente. Para assistentes de prova, que precisam garantir terminação, a definição de tipos indutivos deve obedecer a uma séries de restrições sintáticas. Essas restrições sintáticas visam a garantir que funções que operam sobre esses tipos indutivos, definidas com o uso de recursores, sempre terminam a sua execução. Já para linguagens de programação, que usam recursão geral, não existem limitações na definição de tipos indutivos.

As Figuras 4.1 e 4.2 mostram a implementação da função *dobro* em *Haskell* e em *Coq*. Notar como *Haskell* usa casamento de padrões e utiliza recursão geral, o que faz

com que o código seja muito mais legível para programadores que estão acostumados com essas estruturas. Já *Coq* implementa a função utilizando o recursor, o que à primeira vista pode não ser tão intuitivo, mas garante que o algoritmo sempre termina.

Figura 4.1: Definição da função dobro em Haskell

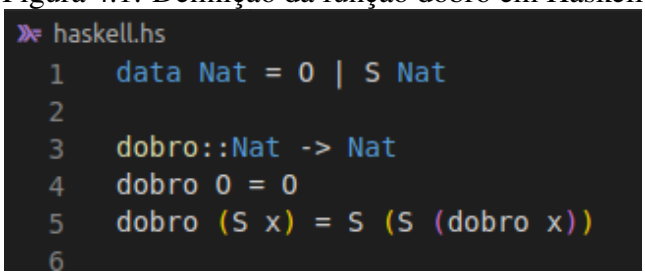
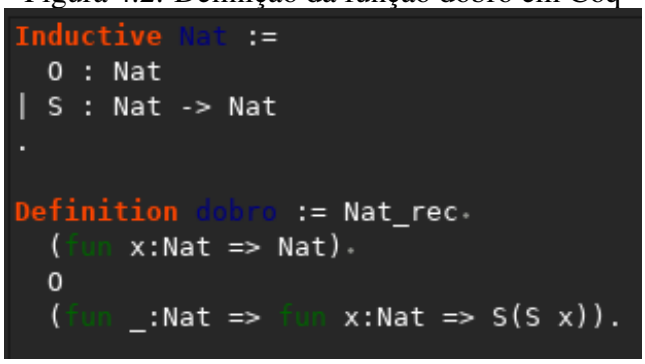


Figura 4.2: Definição da função dobro em Coq



Fonte: Coq IDE

Todos os tipos indutivos são definidos com seus construtores, de forma genérica como em 4.2.

$$\begin{aligned} \tau : * = \ & Construtor_1 : \sigma_1^1 \to \cdots \sigma_m^1 \to \tau \\ & \mid \cdots \\ & \mid \ Construtor_n : \sigma_1^n \to \cdots \sigma_m^n \to \tau \end{aligned} \tag{4.2}$$

Na gramática acima τ é o tipo indutivo sendo definido e σ_j^i é o tipo do parâmetro de índice j do construtor $Construtor_i$.

Para montar um construtor válido no simulador, o tipo indutivo só pode aparecer nos parâmetros dos construtores em posições estritamente positivas, ou seja, se τ existir em σ ele deve estar na saída do tipo. Por exemplo, σ_j^i deve estar no formato $C_1 \to \cdots \to C_n \to X$ onde τ não aparece em C_i e X pode ou não ser igual a τ. Essa restrição não existe em linguagens de programação funcionais, pois a garantia de terminação não é um requisito nessas linguagens.

Uma vez definidos, o simulador precisa de algum mecanismo para disponibilizar os construtores e destrutores ao usuário, com as regras de tipo e semântica específicas. Para isso, é necessária uma nova estrutura de dados, o contexto de definições indutivas, que será usado pelo algorítimo de checagem de tipo e de avaliação.

À gramática simplificada de λC 3.12 foram adicionadas duas novas primitivas, uma para constantes, usadas para identificar nomes dos tipos indutivos e seus construtores, e a primitiva Rec para ser usada como destrutores dos tipos indutivos. Abaixo, segue a gramática atualizada com as novas primitivas destacadas em negrito (4.3).

$$\begin{aligned} M \ ::= \ * \ | \ \Box \ | \ x \ | \ MN \ | \ \lambda x : M.N \ | \ \Pi X : M.N \\ | \ \boldsymbol{Const\ x} \ | \ \mathbf{Rec}\ \boldsymbol{x\ M_1\ M_2\ N_1 \ldots N_n} \end{aligned} \tag{4.3}$$

A Figura 4.3 mostra a implementação de uma função que calcula o fatorial de um natural. Ela foi implementada dentro do simulador, usando as estruturas de tipos indutivos de naturais e pares e utiliza seus construtores e destrutores.

Figura 4.3: Exemplo no simulador - Fatorial usando tipos indutivos

```
inductivedef

  Nat = 0:Nat
      | S:Nat -> Nat;

  Pair = pair: Nat -> Nat -> Pair;

end
let

  fst = \p:Pair. Pair_rec Nat p (\a:Nat.\b:Nat.a);
  snd = \p:Pair. Pair_rec Nat p (\a:Nat.\b:Nat.b);

  soma = \n1:Nat. \n2:Nat. Nat_rec Nat n1 n2 S;
  mult = \n1:Nat. \n2:Nat. Nat_rec Nat n1 0 (\n:Nat. soma n2 n);

  fatCaso0 = (pair 0 (S 0));
  fatCasoS = (\p:Pair. pair (S (fst p)) (mult (S (fst p)) (snd p)));
  fat = \n:Nat. snd (Nat_rec Pair n fatCaso0 fatCasoS);

in

  fat
```

Tipo

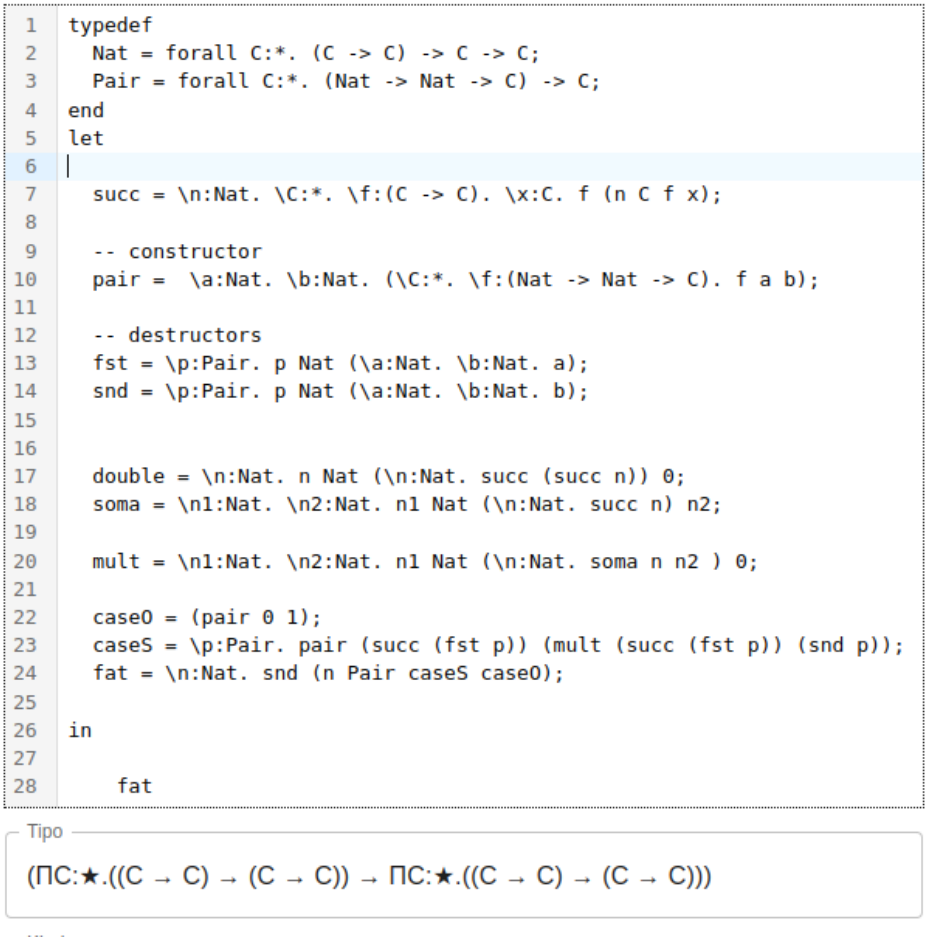
(Nat → Nat)

Fonte: Simulador Cálculo Lambda

Por motivo de comparação, a Figura 4.4 mostra a implementação da mesma função fatorial implementada usando apenas polimorfismo. Notar a diferença entre as definições dos tipos utilizados e o tipo final da função fatorial na parte inferior da imagem.

O tipo da função que utiliza tipos indutivos se mostra mais clara, pois não utiliza tipos codificados.

Figura 4.4: Exemplo no simulador - Fatorial usando polimorfismo



```
typedef
  Nat = forall C:*. (C -> C) -> C -> C;
  Pair = forall C:*. (Nat -> Nat -> C) -> C;
end
let

  succ = \n:Nat. \C:*. \f:(C -> C). \x:C. f (n C f x);

  -- constructor
  pair =  \a:Nat. \b:Nat. (\C:*. \f:(Nat -> Nat -> C). f a b);

  -- destructors
  fst = \p:Pair. p Nat (\a:Nat. \b:Nat. a);
  snd = \p:Pair. p Nat (\a:Nat. \b:Nat. b);


  double = \n:Nat. n Nat (\n:Nat. succ (succ n)) 0;
  soma = \n1:Nat. \n2:Nat. n1 Nat (\n:Nat. succ n) n2;

  mult = \n1:Nat. \n2:Nat. n1 Nat (\n:Nat. soma n n2 ) 0;

  case0 = (pair 0 1);
  caseS = \p:Pair. pair (succ (fst p)) (mult (succ (fst p)) (snd p));
  fat = \n:Nat. snd (n Pair caseS case0);

in

    fat
```

Tipo

(ΠC:★.((C → C) → (C → C)) → ΠC:★.((C → C) → (C → C)))

Fonte: Simulador Cálculo Lambda

Outro exemplo de estrutura mais complexa que pode ser representada com tipos indutivos é a árvore de naturais. A Figura 4.5 mostra a codificação de uma estrutura de dados de árvore de naturais e a implementação de uma função que calcula a soma dos nodos dessa árvore. A mesma implementação sem usar tipos indutivos é possível, mas os tipos das funções tendem a ficar difíceis de ler, pois utilizariam tipos codificados.

Existem ainda diversas extensões que podem ser feitas aos tipos indutivos implementados no simulador, como tipos indutivos parametrizáveis que recebem parâmetros, permitindo, por exemplo, a definição de um tipo indutivo de lista cujos elementos são de um tipo τ, definido abaixo (4.4).

$$\forall\tau.List\ \tau = empty : List\ \tau \mid cons : \tau \rightarrow List\ \tau \tag{4.4}$$

Figura 4.5: Exemplo no simulador - Função que soma os nodos de uma árvore usando tipos indutivos

```
inductivedef

  Nat = O:Nat
      | S:Nat -> Nat;

  Tree = Null:Tree
       | Node: Nat -> Tree -> Tree -> Tree;

end
let

  soma = \n1:Nat. \n2:Nat. Nat_rec Nat n1 n2 S;

  somaTree = \t:Tree. Tree_rec Nat t O
             (\n:Nat.\r1:Nat.\r2:Nat. soma n (soma r1 r2));

  tree1 = Node (S (S (S O)))
            (Node (S O) Null Null)
            (Node (S (S O)) Null Null);

in
   somaTree tree1
```

Fonte: Simulador Cálculo Lambda

5 FERRAMENTA

Esse capítulo tem o objetivo de explorar a interface e todas as funcionalidades do simulador com exemplos e passando por alguns dos vértices do cálculo lambda e outras extensões. O código da ferramenta pode ser encontrada em <https://bitbucket.org/ebchandelier/lambda-calculus-simulator/src/master/> e a ferramenta está acessível em <https://www.inf.ufrgs.br/~ebchandelier/>.

5.1 Simulador

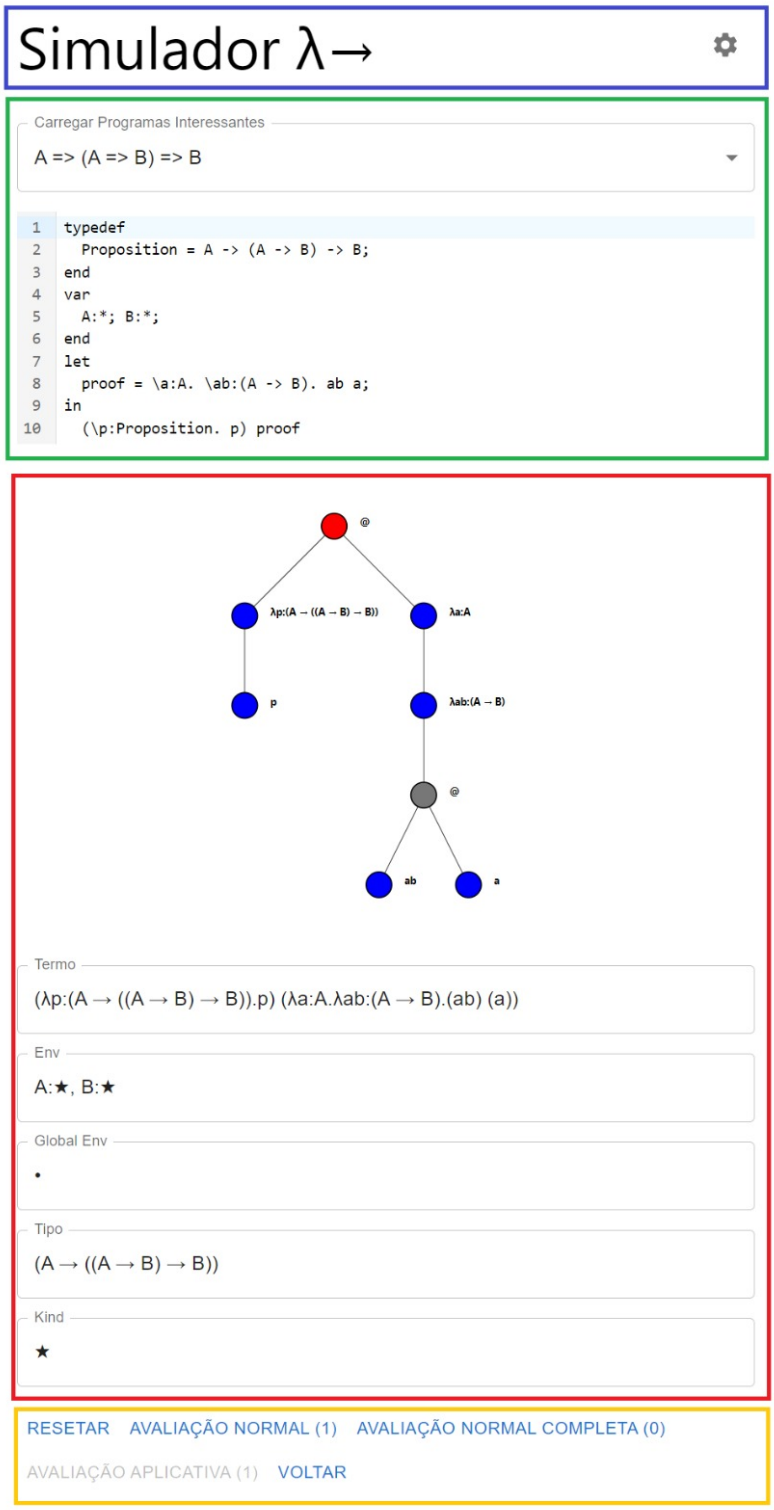
A interface inicial do simulador possui diversos componentes de interação com o usuário, destacados com cores na Figura 5.1. Marcado em azul está o nome do cálculo para o qual ele está configurado, ao lado do nome está o botão para abrir as configurações do simulador. Em verde está o editor e um utilitário para carregar programas iniciais. Em vermelho está o termo representado em forma de árvore, o termo representado como texto, o tipo do termo, o $kind$ do tipo, e os dois contextos, o de declaração de variáveis e o de definições globais. E por último, em amarelo, uma série de botões para avaliação do termo.

5.1.1 Configuração do simulador

Na modal de configurações do simulador (ver Figura 5.2) podemos escolher em qual vértice do cubo lambda vamos trabalhar, escolhendo entre ativar os recursos de polimorfismo, construção de tipos e tipos dependentes. O simulador é expandido com tipos indutivos por padrão. Todas as outras configurações desabilitadas ainda não foram desenvolvidas.

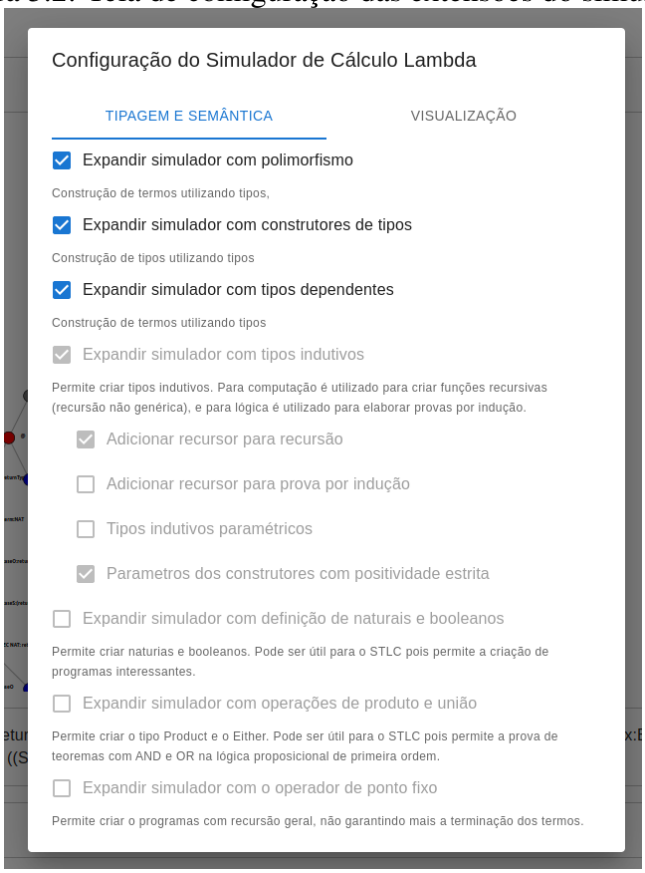
Ainda na modal de configurações na tab Visualização (Figura 5.3), pode-se configurar quais componentes de layout serão renderizados na tela. Existem diversos componentes que podem ser úteis para diferentes tipos de cálculos, então podemos escolher quais vão estar visíveis dependendo de qual exercício se está tentando resolver com o simulador.

Figura 5.1: Interface do Simulador com todos os componentes



Fonte: Simulador Cálculo Lambda

Figura 5.2: Tela de configuração das extensões do simulador



Fonte: Simulador Cálculo Lambda

5.1.2 Sintaxe

O código do simulador é dividido em cinco blocos (Figura 5.4): Os tipos indutivos são definidas dentro do bloco $inductivedef\ \cdots\ end$; os tipos dentro do bloco $typedef\ \cdots\ end$; as variáveis iniciais do ambiente dentro do bloco $var\ \cdots\ end$; e as definições de funções dentro do bloco $let\ \cdots\ in$.

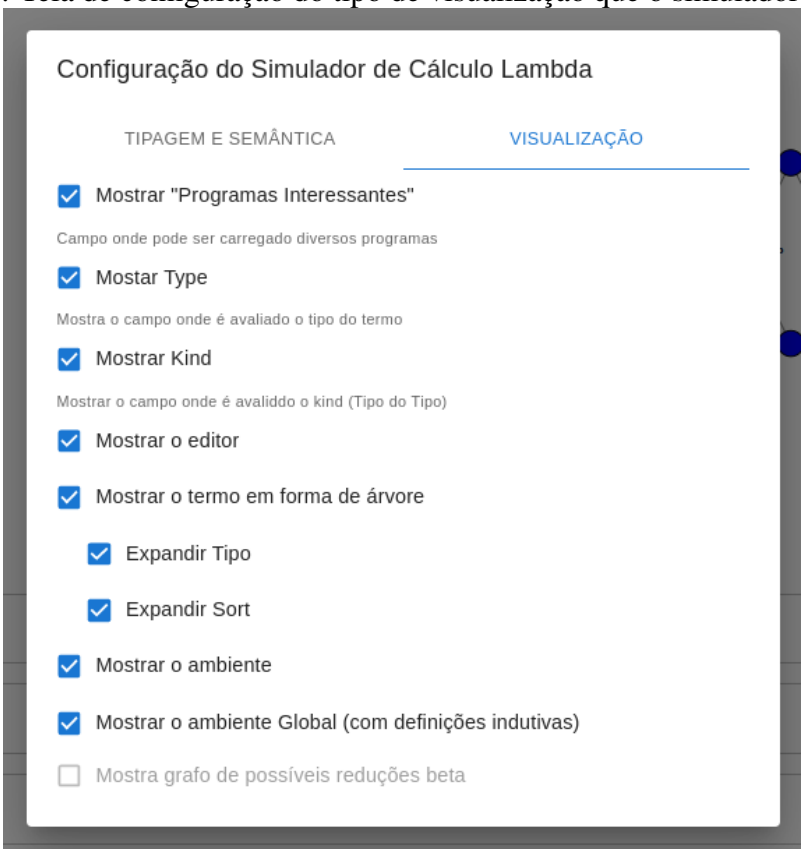
O último bloco é o que segue depois da palavra in e ele terá suas variáveis substituídas por todas as definições dos blocos anteriores.

Com a exceção do bloco com o termo principal, todos os blocos podem ter diversas declarações, separadas com ";".

A seguir serão mostrados elementos de sintaxe concreta necessários para a edição de termos sintaticamente válidos no simulador.

- Variáveis - Qualquer palavra que comece com uma letra vai ser interpretada como um variável, salvo as palavras reservadas do sistema, como let, in, var, $typedef$, end, $forall$ e $inductivedef$.

Figura 5.3: Tela de configuração do tipo de visualização que o simulador vai exibir.



Fonte: Simulador Cálculo Lambda

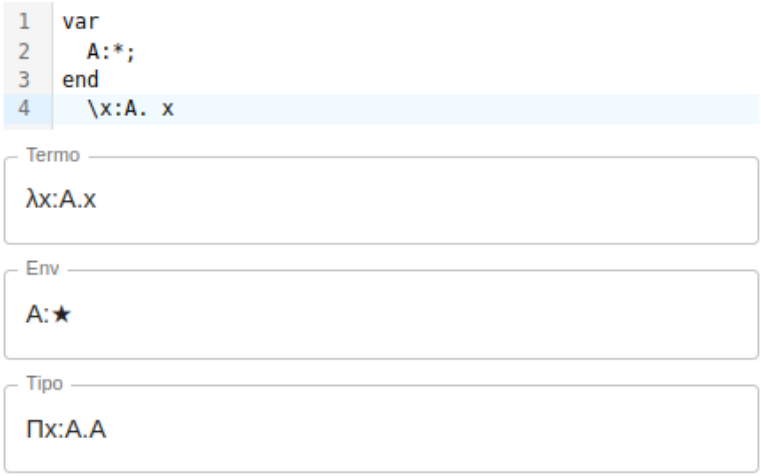
Figura 5.4: Sintaxe do simulador - Estrutura geral

```
1  inductivedef
2    -- inductive type definitions...
3  end
4  typedef
5    -- type definitions
6  end
7  var
8    -- context definitions...
9  end
10 let
11   -- terms definition...
12 in
13   x -- main term
```

Fonte: Simulador Cálculo Lambda

- Abstrações Lambda - Para definir um termo $\lambda x : A.x$ no simulador usamos a sintaxe $\backslash x : A.x$ como em 5.5. Note que precisamos definir o tipo A no bloco var para essa expressão estar bem tipada, Observe na Figura 5.5 que no campo Ambiente temos que A:* e no campo Tipo temos o tipo da abstração lambda que é $\Pi x : A.A$.
- Π Type - Para definir, por exemplo, o termo $\Pi x : A.A$, usamos a sintaxe $forall\ x : A.A$ ou $A -> A$ simplesmente, uma vez que x não ocorre em A. Notar que as

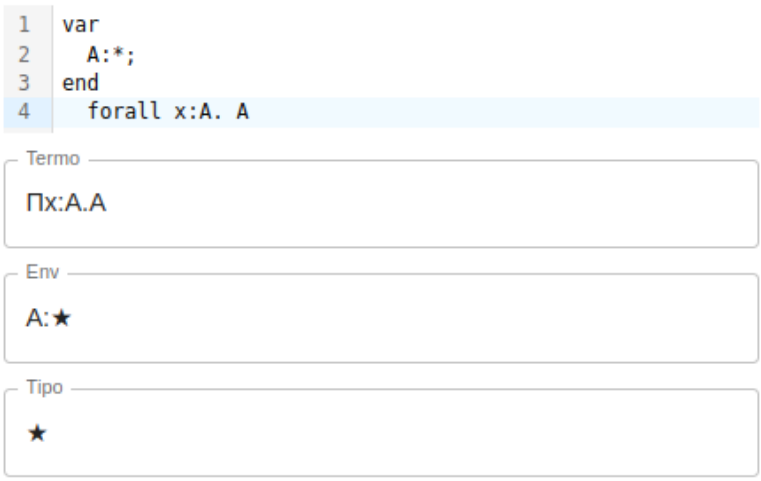
Figura 5.5: Sintaxe do simulador - Definição de expressões Lambda



Fonte: Simulador Cálculo Lambda

variáveis de tipo precisam estar definidas no contexto para a expressão estar bem tipada, e, como esperado, o tipo de um tipo, ou $Kind$, é $*$ e pode ser visto no campo Tipo da figura 5.6.

Figura 5.6: Sintaxe do simulador - Definição de expressões Lambda

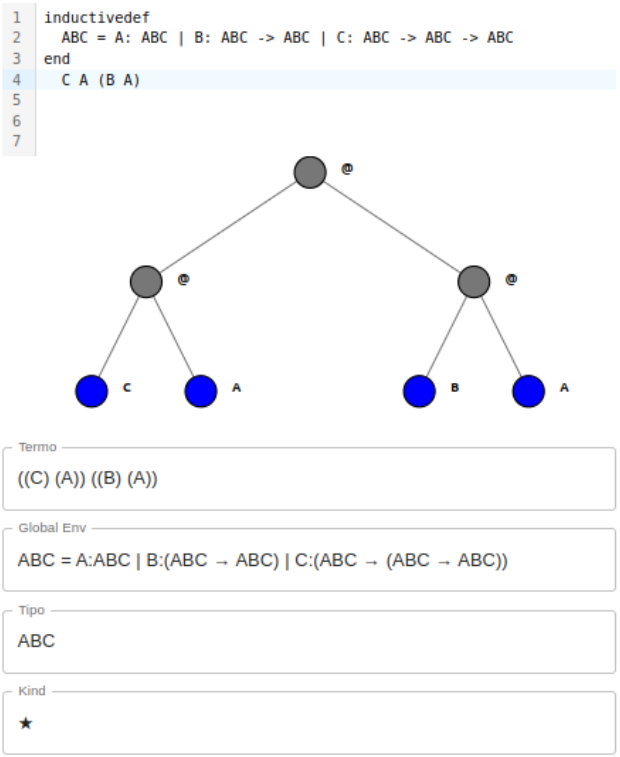


Fonte: Simulador Cálculo Lambda

- Tipo Indutivo - Para cada novo tipo indutivo, definimos seu nome, o nome de seus construtores e seus tipos. Um exemplo de definição de um tipo indutivo aparece na Figura 5.7.

Para fins didáticos, o simulador apresenta o termo em forma de árvore. A imagem é atualizada de forma dinâmica ao editar o código. Para visualizar a árvore basta entrar na tela de configurações na Tab de Visualização e marcar "Mostrar Termo em Forma de Árvore". Os *redexes* são pintados em vermelho para indicar possíveis Beta Reduções,

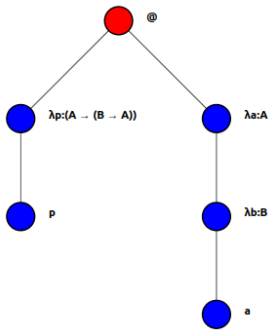
Figura 5.7: Sintaxe do simulador - Definição do Tipo Indutivo ABC, com 3 construtores



Fonte: Simulador Cálculo Lambda

como em 5.8.

Figura 5.8: Termo em forma de árvore mostrada pelo simulador de cálculo lambda, a figura mostra o termo $A \to B \to A$

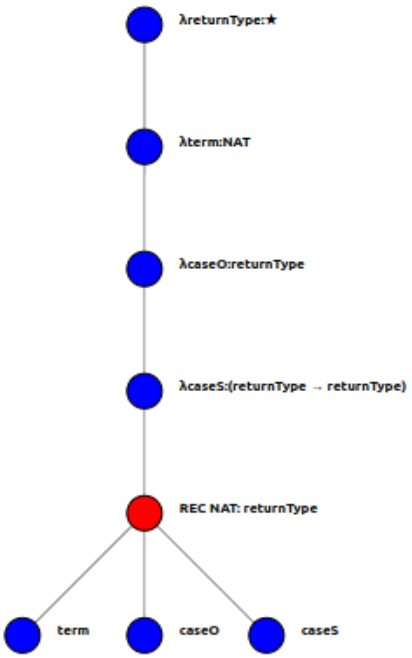


Fonte: Simulador Cálculo Lambda

A Figura 5.9 mostra a estrutura da árvore para o termo $\mathrm{Rec}_{\mathrm{Nat}}$. É possível ver todos os parâmetros necessários para montar o recursor, em especial o primeiro parâmetro representado pela abstração lambda $\lambda returnType : * \ldots$ para definir qual é o tipo de saída desse recursor. Por esse detalhe de implementação, é necessário o uso de polimorfismo para utilizar um recursor no simulador.

Os tipos dos termos são mostrados por padrão contraídos, mas como tipos também podem ter *redexes*, pode-se habilitar a configuração "Expandir Tipo"para ver esse tipo

Figura 5.9: Exemplo no simulador - Recursor dos Naturais



Fonte: Simulador Cálculo Lambda

expandido e todos os seus *redexes*. De forma análoga, é possível visualizar os $Sorts$, tipos dos tipos, de forma expandida habilitando a opção "Expandir Sort".

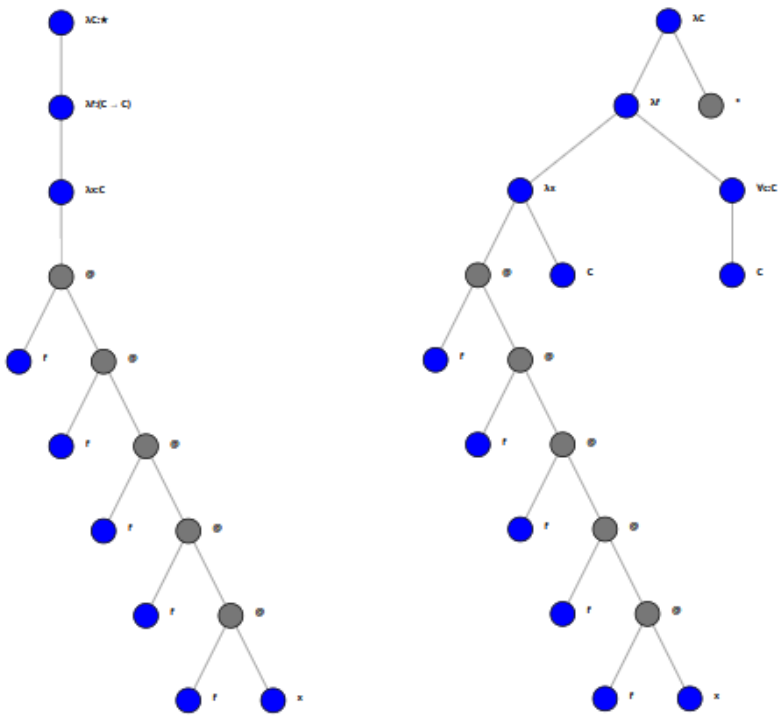
A Figura 5.10 mostra o mesmo termo sem expandir o tipo e expandindo o tipo. Contrair a parte do termo que indica o tipo pode ser útil para quem está usando o simulador para ter uma visão abstrata de certas partes do termo.

Por último, na parte inferior da tela da ferramenta, estão os botões para avaliar o termo. Ao avaliar um termo o simulador mantém um histórico de termos passados, para que se possa desfazer reduções.

5.2 Arquitetura e Tecnologias

Para modelar as principais estruturas de dados e os principais algoritmos usados na implementação do simulador foi escolhida a linguagem de programação $Purescript$ (Phil Freeman, 2013), pois suporta tipos indutivos e casamento de padrões, tornando mais direta a tradução de definições formais em código. Além disso, é uma linguagem que compila para *Javascript*, permitindo que o simulador rode diretamente no navegador e,

Figura 5.10: Comparação do mesmo termo em forma de árvore. Na forma contraída, com os tipos expandidos, e os *Sorts* expandidos.



Fonte: Simulador Cálculo Lambda

portanto, tornando-o acessível a partir de uma grande variedade de dispositivos.

Para a visualização no navegador foi utilizado *ReactJs*, um framework que possibilita misturar HTML, CSS e *Javascript* para construir páginas Web dinâmicas, ou seja, que tem o seu conteúdo modificado dinamicamente com a interação do usuário. Nele foram utilizadas diversas bibliotecas, como *@mui/material* para os componentes de layout, *react-final-form* para controlar as configurações do simulador e a edição do código, *react-codemirror* para receber o *input* de código do usuário e *react-d3-tree* para renderizar o termo em forma de árvore. As versões das bibliotecas utilizadas podem ser vistas no arquivo *package.json* no repositório.

A compilação do simulador se dá em duas etapas. Na primeira etapa é feita a compilação de código *PureScript* para *JavaScript*, que será usada na próxima etapa. Na segunda etapa é feita a compilação dos componentes de *layout* com o código gerado na etapa anterior.

A cada edição do código, todos os componentes de *layout* são atualizados de forma reativa, ou seja, sobre demanda. A página Web pode ser vista como uma árvore de componentes, onde cada um dos componentes recebem parâmetros do pai, e manda

parâmetros para os filhos, o Framework ReactJS sabe então quando é necessário atualizar e renderizar algum componente, o que faz com que os usuários tenham uma experiência melhor ao utilizar o simulador. No simulador existe o estado para o código de entrada e as configurações de tipagem e visualização. A cada edição do código, o primeiro algoritmo que é executado é o *parser*, que lê o código e devolve o termo, o ambiente inicial e o ambiente global que ele representa. Após, o campo "Tipo"é atualizado com o algoritmo de tipagem, que utiliza como parâmetros o termo, os dois ambiente e as estados do simulador que indicam a presença de polimorfismo, construtores de tipos e tipos dependentes.

6 CONCLUSÃO

Este trabalho cumpre o objetivo de apresentar uma ferramenta para substituir as quatro já utilizadas na disciplina de Cálculo Lambda e Teoria de Tipos (INF05013). Nela é possível programar em qualquer um dos vértices do Cubo Lambda com uma sintaxe única, diferente das ferramentas anteriores com as quais só era possível programar nos vértices dos cálculos $\lambda_{\rightarrow}$, $\lambda 2$, $\lambda\omega$ e λC com uma sintaxe específica para cada cálculo. Além disso, foi incorporada na ferramenta a possibilidade de definir e utilizar Tipos Indutivos, não existente nos simuladores anteriores. A ferramenta pode ser encontrada no link <https: //www.inf.ufrgs.br/~ebchandelier/>.

Diversas melhorias podem ser feitas na ferramenta. Do ponto de vista dos cálculos do Cubo Lambda podem ser feitas melhorias como permitir a extensão do simulador com diversas primitivas como naturais, booleanos e operadores de lógica. Do ponto de vista de tipos indutivos, podem ser feitas melhorias como: adicionar uma regra nova de tipagem do recursor Rec que permita a realização de provas por indução, pois o estado atual do simulador ainda não implementa essa funcionalidade; uma implementação diferente do recursor Rec para possibilitar a tipagem desse termo independente de polimorfismo; e a implementação de tipos indutivos paramétricos para possibilitar a construção de tipos indutivos com parâmetros. Do ponto de vista de visualização e didática do simulador, podem ser adicionados recursos como, por exemplo: a possibilidade de visualizar a árvore de prova para facilitar os exercícios de provas de fórmulas da lógica, e a visualização de um grafo de possíveis Beta Reduções para estudar diferentes estratégias de avaliação.

REFERÊNCIAS

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

BERTOT, Y.; CASTÉRAN, P. **Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions**. [S.l.]: Springer Science & Business Media, 2013.

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**, Cambridge University Press, v. 5, n. 2, p. 56–68, 1940.

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>.

GIRARD, J.-Y. **Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur**. Thesis (Thèse d'État) — Université Paris VII, 1972.

GöDEL, V. K. Über eine bisher noch nicht benÜtzte erweiterung des finiten standpunktes. **Dialectica**, v. 12, n. 3-4, p. 280–287, 1958. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1746-8361.1958.tb01464.x>.

MOURA, L. M. de et al. The lean theorem prover (system description). In: FELTY, A. P.; MIDDELDORP, A. (Ed.). **CADE**. Springer, 2015. (Lecture Notes in Computer Science, v. 9195), p. 378–388. ISBN 978-3-319-21400-9. Available from Internet: <http://dblp.uni-trier.de/db/conf/cade/cade2015.html#MouraKADR15>.

Phil Freeman. **Purescript**. 2013. Available from Internet: <https://www.purescript.org/>.

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>.

REYNOLDS, J. C. Towards a theory of type structure. In: ROBINET, B. (Ed.). **Programming Symposium**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974. p. 408–425. ISBN 978-3-540-37819-8.

RUSSELL, B. Mathematical logic as based on the theory of types. **American Journal of Mathematics**, v. 30, p. 222, 1908.

SøRENSEN, M. H.; URZYCZYN, P. **Lectures on the Curry-Howard isomorphism**. Amsterdam; Oxford: Elsevier, 2006. ISBN 0444520775 9780444520777. Available from Internet: <https://www.worldcat.org/title/lectures-on-the-curry-howard-isomorphism/oclc/1171193011&referer=brief_results>.

TURING, A. M. On computable numbers, with an application to the Entscheidungsproblem. **Proceedings of the London Mathematical Society**, v. 2, n. 42, p. 230–265, 1936. Available from Internet: <http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf>.

WELLS, J. Typability and type checking in system f are equivalent and undecidable. **Annals of Pure and Applied Logic**, v. 98, n. 1, p. 111–156, 1999. ISSN 0168-0072. Available from Internet: <https://www.sciencedirect.com/science/article/pii/S0168007298000475>.