TURNO:	**NOTURNO**	VERSÃO:	1	ANO / SEMESTRE:	2014.2	Nº	

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO — BACHARELADO
COORDENAÇÃO DE TRABALHO DE CONCLUSÃO DE CURSO

PROPOSTA PARA O TRABALHO DE CONCLUSÃO DE CURSO

TÍTULO: PROVADOR INTERATIVO DE TEOREMAS COM TIPOS DEPENDENTES

ÁREA: Teoria dos Tipos

Palavras-chave: Assistente de provas. Linguagens de programação. Sistemas de tipos. Teoria dos tipos dependentes.

1 IDENTIFICAÇÃO

1.1 ALUNO

Nome: **André Ramaciotti da Silva**			Código/matrícula: **60239**	
Endereço residencial:				
Rua: **Prudente de Moraes**			n°: **378**	Complemento: **Apto 904**
Bairro: **Vila Nova**	CEP: **89035-360**	Cidade: **Blumenau**		UF: **SC**
Telefone fixo: **(47) 3234-0962**		Celular: **(47) 9609-8990**		
Endereço comercial:				
Empresa: **Teclógica Sistemas em Informática Ltda.**				
Rua: **XV de Novembro**			n°: **759**	Bairro: **Centro**
CEP: **89010-902**	Cidade: **Blumenau**		UF: **SC**	Telefone: **(47) 3036-7700**
E-Mail FURB:		E-Mail alternativo: **andre.ramaciotti@gmail.com**		

1.2 ORIENTADOR

Nome: **Joyce Martins**	
E-Mail FURB: **joyce@furb.br**	E-Mail alternativo: **joyyce.martins@gmail.com**

2 DECLARAÇÕES

2.1 DECLARAÇÃO DO ALUNO

Declaro que estou ciente do Regulamento do Trabalho de Conclusão de Curso de Ciência da Computação e que a proposta em anexo, a qual concordo, foi por mim rubricada em todas as páginas. Ainda me comprometo pela obtenção de quaisquer recursos necessários para o desenvolvimento do trabalho, caso esses recursos não sejam disponibilizados pela Universidade Regional de Blumenau (FURB).

Assinatura: _____ Local/data: _____

2.2 DECLARAÇÃO DO ORIENTADOR

Declaro que estou ciente do Regulamento do Trabalho de Conclusão do Curso de Ciência da Computação e que a proposta em anexo, a qual concordo, foi por mim rubricada em todas as páginas. Ainda me comprometo a orientar o aluno da melhor forma possível de acordo com o plano de trabalho explícito nessa proposta.

Assinatura: _____ Local/data: _____

3 AVALIAÇÃO DA PROPOSTA

3.1 AVALIAÇÃO DO(A) **ORIENTADOR(A)**

Acadêmico(a): André Ramaciotti da Silva

Orientador(a): Joyce Martins

	ASPECTOS AVALIADOS	atende	atende parcialmente	não atende
ASPECTOS TÉCNICOS	1. INTRODUÇÃO 1.1. O tema de pesquisa está devidamente contextualizado/delimitado?			
	1.2. O problema está claramente formulado?			
	2. OBJETIVOS 2.1. O objetivo geral está claramente definido e é passível de ser alcançado?			
	2.2. São apresentados objetivos específicos (opcionais) coerentes com o objetivo geral? Caso não sejam apresentados objetivos específicos, deixe esse item em branco.			
	3. RELEVÂNCIA 3.1. A proposta apresenta um grau de relevância em computação que justifique o desenvolvimento do TCC?			
	4. METODOLOGIA 4.1. Foram relacionadas todas as etapas necessárias para o desenvolvimento do TCC?			
	4.2. Os métodos e recursos estão devidamente descritos e são compatíveis com a metodologia proposta?			
	4.3. A proposta apresenta um cronograma físico (período de realização das etapas) de maneira a permitir a execução do TCC no prazo disponível?			
	5. REVISÃO BIBLIOGRÁFICA 5.1. As informações apresentadas são suficientes e têm relação com o tema do TCC?			
	5.2. São apresentados trabalhos correlatos, bem como comentadas as principais características dos mesmos?			
	6. REQUISITOS DO SISTEMA A SER DESENVOLVIDO 6.1. Os requisitos funcionais e não funcionais do sistema a ser desenvolvido foram claramente descritos?			
	7. CONSIDERAÇÕES FINAIS 7.1. As considerações finais relacionam os assuntos apresentados na revisão bibliográfica com a realização do TCC?			
ASPECTOS METODOLÓGICOS	8. REFERÊNCIAS BIBLIOGRÁFICAS 8.1. As referências bibliográficas obedecem às normas da ABNT?			
	8.2. As referências bibliográficas contemplam adequadamente os assuntos abordados na proposta (são usadas obras atualizadas e/ou as mais importantes da área)?			
	9. CITAÇÕES 9.1. As citações obedecem às normas da ABNT?			
	9.2. As informações retiradas de outros autores estão devidamente citadas?			
	10. AVALIAÇÃO GERAL (organização e apresentação gráfica, linguagem usada) 10.1. O texto obedece ao formato estabelecido?			
	10.2. A exposição do assunto é ordenada (as idéias estão bem encadeadas e a linguagem utilizada é clara)?			

A proposta de TCC deverá ser revisada, isto é, necessita de complementação, se:

- qualquer um dos itens tiver resposta NÃO ATENDE;
- pelo menos **4 (quatro)** itens dos **ASPECTOS TÉCNICOS** tiverem resposta ATENDE PARCIALMENTE; ou
- pelo menos **4 (quatro)** itens dos **ASPECTOS METODOLÓGICOS** tiverem resposta ATENDE PARCIALMENTE.

PARECER: ( ) APROVADA ( ) NECESSITA DE COMPLEMENTAÇÃO

Assinatura do(a) avaliador(a): _____ Local/data: _____

CONSIDERAÇÕES DO(A) ORIENTADOR(A):

Caso o(a) orientador(a) tenha assinalado em sua avaliação algum item como “atende parcialmente”, devem ser relatos os problemas/melhorias a serem efetuadas.

Na segunda versão, caso as alterações sugeridas pelos avaliadores não sejam efetuadas, deve-se incluir uma justificativa.

Assinatura do(a) avaliador(a): _____ Local/data: _____

3.2 AVALIAÇÃO/HOMOLOGAÇÃO DO **COORDENADOR DE TCC**

Acadêmico(a): André Ramaciotti da Silva

Avaliador(a): Maurício Capobianco Lopes

	ASPECTOS AVALIADOS	atende	atende parcialmente	não atende
ASPECTOS TÉCNICOS	1. INTRODUÇÃO 1.1. O tema de pesquisa está devidamente contextualizado/delimitado?			
	1.2. O problema está claramente formulado?			
	2. OBJETIVOS 2.1. O objetivo geral está claramente definido e é passível de ser alcançado?			
	2.2. São apresentados objetivos específicos (opcionais) coerentes com o objetivo geral? Caso não sejam apresentados objetivos específicos, deixe esse item em branco.			
	3. RELEVÂNCIA 3.1. A proposta apresenta um grau de relevância em computação que justifique o desenvolvimento do TCC?			
	4. METODOLOGIA 4.1. Foram relacionadas todas as etapas necessárias para o desenvolvimento do TCC?			
	4.2. Os métodos e recursos estão devidamente descritos e são compatíveis com a metodologia proposta?			
	4.3. A proposta apresenta um cronograma físico (período de realização das etapas) de maneira a permitir a execução do TCC no prazo disponível?			
	5. REVISÃO BIBLIOGRÁFICA 5.1. As informações apresentadas são suficientes e têm relação com o tema do TCC?			
	5.2. São apresentados trabalhos correlatos, bem como comentadas as principais características dos mesmos?			
	6. REQUISITOS DO SISTEMA A SER DESENVOLVIDO 6.1. Os requisitos funcionais e não funcionais do sistema a ser desenvolvido foram claramente descritos?			
	7. CONSIDERAÇÕES FINAIS 7.1. As considerações finais relacionam os assuntos apresentados na revisão bibliográfica com a realização do TCC?			
ASPECTOS METODOLÓGICOS	8. REFERÊNCIAS BIBLIOGRÁFICAS 8.1. As referências bibliográficas obedecem às normas da ABNT?			
	8.2. As referências bibliográficas contemplam adequadamente os assuntos abordados na proposta (são usadas obras atualizadas e/ou as mais importantes da área)?			
	9. CITAÇÕES 9.1. As citações obedecem às normas da ABNT?			
	9.2. As informações retiradas de outros autores estão devidamente citadas?			
	10. AVALIAÇÃO GERAL (organização e apresentação gráfica, linguagem usada) 10.1. O texto obedece ao formato estabelecido?			
	10.2. A exposição do assunto é ordenada (as idéias estão bem encadeadas e a linguagem utilizada é clara)?			

A proposta de TCC deverá ser revisada, isto é, necessita de complementação, se:

- qualquer um dos itens tiver resposta NÃO ATENDE;
- pelo menos **4 (quatro)** itens dos **ASPECTOS TÉCNICOS** tiverem resposta ATENDE PARCIALMENTE; ou
- pelo menos **4 (quatro)** itens dos **ASPECTOS METODOLÓGICOS** tiverem resposta ATENDE PARCIALMENTE.

PARECER: ( ) APROVADA ( ) NECESSITA DE COMPLEMENTAÇÃO

OBSERVAÇÕES:

Assinatura do(a) avaliador(a): _____ Local/data: _____

3.3 AVALIAÇÃO DO **PROFESSOR DA DISCIPLINA DE TCCI**

Acadêmico(a): André Ramaciotti da Silva

Avaliador(a): Roberto Heinzle

	ASPECTOS AVALIADOS	atende	atende parcialmente	não atende
ASPECTOS TÉCNICOS	1. INTRODUÇÃO 1.1. O tema de pesquisa está devidamente contextualizado/delimitado?			
	1.2. O problema está claramente formulado?			
	2. OBJETIVOS 2.1. O objetivo geral está claramente definido e é passível de ser alcançado?			
	2.2. São apresentados objetivos específicos (opcionais) coerentes com o objetivo geral? Caso não sejam apresentados objetivos específicos, deixe esse item em branco.			
	3. RELEVÂNCIA 3.1. A proposta apresenta um grau de relevância em computação que justifique o desenvolvimento do TCC?			
	4. METODOLOGIA 4.1. Foram relacionadas todas as etapas necessárias para o desenvolvimento do TCC?			
	4.2. Os métodos e recursos estão devidamente descritos e são compatíveis com a metodologia proposta?			
	4.3. A proposta apresenta um cronograma físico (período de realização das etapas) de maneira a permitir a execução do TCC no prazo disponível?			
	5. REVISÃO BIBLIOGRÁFICA 5.1. As informações apresentadas são suficientes e têm relação com o tema do TCC?			
	5.2. São apresentados trabalhos correlatos, bem como comentadas as principais características dos mesmos?			
	6. REQUISITOS DO SISTEMA A SER DESENVOLVIDO 6.1. Os requisitos funcionais e não funcionais do sistema a ser desenvolvido foram claramente descritos?			
	7. CONSIDERAÇÕES FINAIS 7.1. As considerações finais relacionam os assuntos apresentados na revisão bibliográfica com a realização do TCC?			
ASPECTOS METODOLÓGICOS	8. REFERÊNCIAS BIBLIOGRÁFICAS 8.1. As referências bibliográficas obedecem às normas da ABNT?			
	8.2. As referências bibliográficas contemplam adequadamente os assuntos abordados na proposta (são usadas obras atualizadas e/ou as mais importantes da área)?			
	9. CITAÇÕES 9.1. As citações obedecem às normas da ABNT?			
	9.2. As informações retiradas de outros autores estão devidamente citadas?			
	10. AVALIAÇÃO GERAL (organização e apresentação gráfica, linguagem usada) 10.1. O texto obedece ao formato estabelecido?			
	10.2. A exposição do assunto é ordenada (as idéias estão bem encadeadas e a linguagem utilizada é clara)?			
	PONTUALIDADE NA ENTREGA		atraso de _____ dias	

A proposta de TCC deverá ser revisada, isto é, necessita de complementação, se:

- qualquer um dos itens tiver resposta NÃO ATENDE;
- pelo menos **4 (quatro)** itens dos **ASPECTOS TÉCNICOS** tiverem resposta ATENDE PARCIALMENTE; ou
- pelo menos **4 (quatro)** itens dos **ASPECTOS METODOLÓGICOS** tiverem resposta ATENDE PARCIALMENTE.

PARECER: ( ) APROVADA ( ) NECESSITA DE COMPLEMENTAÇÃO

OBSERVAÇÕES:

Assinatura do(a) avaliador(a): _____ Local/data: _____

3.4 AVALIAÇÃO DO(A) **PROFESSOR(A) ESPECIALISTA NA ÁREA**

Acadêmico(a): André Ramaciotti da Silva

Avaliador(a): _____

	ASPECTOS AVALIADOS	atende	atende parcialmente	não atende
ASPECTOS TÉCNICOS	1. INTRODUÇÃO 1.1. O tema de pesquisa está devidamente contextualizado/delimitado?			
	1.2. O problema está claramente formulado?			
	2. OBJETIVOS 2.1. O objetivo geral está claramente definido e é passível de ser alcançado?			
	2.2. São apresentados objetivos específicos (opcionais) coerentes com o objetivo geral? Caso não sejam apresentados objetivos específicos, deixe esse item em branco.			
	3. RELEVÂNCIA 3.1. A proposta apresenta um grau de relevância em computação que justifique o desenvolvimento do TCC?			
	4. METODOLOGIA 4.1. Foram relacionadas todas as etapas necessárias para o desenvolvimento do TCC?			
	4.2. Os métodos e recursos estão devidamente descritos e são compatíveis com a metodologia proposta?			
	4.3. A proposta apresenta um cronograma físico (período de realização das etapas) de maneira a permitir a execução do TCC no prazo disponível?			
	5. REVISÃO BIBLIOGRÁFICA 5.1. As informações apresentadas são suficientes e têm relação com o tema do TCC?			
	5.2. São apresentados trabalhos correlatos, bem como comentadas as principais características dos mesmos?			
	6. REQUISITOS DO SISTEMA A SER DESENVOLVIDO 6.1. Os requisitos funcionais e não funcionais do sistema a ser desenvolvido foram claramente descritos?			
	7. CONSIDERAÇÕES FINAIS 7.1. As considerações finais relacionam os assuntos apresentados na revisão bibliográfica com a realização do TCC?			
ASPECTOS METODOLÓGICOS	8. REFERÊNCIAS BIBLIOGRÁFICAS 8.1. As referências bibliográficas obedecem às normas da ABNT?			
	8.2. As referências bibliográficas contemplam adequadamente os assuntos abordados na proposta (são usadas obras atualizadas e/ou as mais importantes da área)?			
	9. CITAÇÕES 9.1. As citações obedecem às normas da ABNT?			
	9.2. As informações retiradas de outros autores estão devidamente citadas?			
	10. AVALIAÇÃO GERAL (organização e apresentação gráfica, linguagem usada) 10.1. O texto obedece ao formato estabelecido?			
	10.2. A exposição do assunto é ordenada (as idéias estão bem encadeadas e a linguagem utilizada é clara)?			

A proposta de TCC deverá ser revisada, isto é, necessita de complementação, se:

- qualquer um dos itens tiver resposta NÃO ATENDE;
- pelo menos **4 (quatro)** itens dos **ASPECTOS TÉCNICOS** tiverem resposta ATENDE PARCIALMENTE; ou
- pelo menos **4 (quatro)** itens dos **ASPECTOS METODOLÓGICOS** tiverem resposta ATENDE PARCIALMENTE.

PARECER: ( ) APROVADA ( ) NECESSITA DE COMPLEMENTAÇÃO

OBSERVAÇÕES:

Assinatura do(a) avaliador(a): _____ Local/data: _____

UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

PROVADOR INTERATIVO DE TEOREMAS COM TIPOS DEPENDENTES

ANDRÉ RAMACIOTTI DA SILVA

BLUMENAU
2014

ANDRÉ RAMACIOTTI DA SILVA

PROVADOR INTERATIVO DE TEOREMAS COM TIPOS DEPENDENTES

Proposta de Trabalho de Conclusão de Curso submetida à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso I do curso de Ciência da Computação — Bacharelado.

Profa. Joyce Martins - Orientadora

BLUMENAU
2014

1 INTRODUÇÃO

Embora ainda seja um assunto relativamente polêmico dentro da Matemática, matemáticos estão aos poucos aceitando o uso de ferramentas computacionais para auxílio na validação e no desenvolvimento de provas formais (AVIGAD; HARRISON, 2014). Provavelmente, um dos casos mais importantes nessa área foi o teorema das quatro cores[1], comprovado por Appel e Haken (1977), primeiro teorema a ser provado com a ajuda de um computador.

Hudak e Jones (1994) afirmam que, ao mesmo tempo, programadores vêm percebendo as vantagens de se usar linguagens de programação funcionais com sistemas de tipos mais expressivos. Esses sistemas podem ser estendidos com diferentes abstrações e uma que tem recebido destaque recentemente é a teoria de tipos dependentes (MCBRIDE; MCKINNA, 2004), em que o tipo de um termo pode depender do valor de outro termo.

Os avanços das duas áreas se encontram nos provadores interativos de teoremas. Essas ferramentas, entre as quais a mais conhecida possivelmente seja Coq (THE COQ DEVELOPMENT TEAM, 2014), cumprem o papel duplo de verificar definições e teoremas matemáticos, além de certificarem que programas estão corretos com regras de validação mais restritas que a maioria das linguagens de programação utiliza.

Diante desse contexto, percebe-se a importância de um provador interativo de teoremas. Sendo assim, propõe-se o desenvolvimento de um sistema em que programas e teoremas sejam validados e executados de acordo com as regras da teoria de tipos dependentes.

1.1 OBJETIVOS DO TRABALHO

Este trabalho objetiva criar um sistema para auxiliar no desenvolvimento de provas formais através do uso do computador.

Os objetivos específicos são:

a) especificar uma linguagem para descrever programas e teoremas de lógica de primeira ordem;
b) especificar um sistema de tipos para essa linguagem, baseado na teoria de tipos dependentes;
c) desenvolver um interpretador de linha de comando para essa linguagem;

[1] O teorema das quatro cores é um teorema dentro da teoria de grafos que afirma que um mapa plano pode ser colorido com apenas quatro cores sem que regiões vizinhas compartilhem a mesma cor. Foi demonstrado pela primeira vez por Appel e Haken (1977) com o auxílio de um computador.

d) validar os programas e os teoremas de lógica de primeira ordem escritos nessa linguagem.

1.2 RELEVÂNCIA DO TRABALHO

A teoria dos tipos dependentes ainda é relativamente pouco explorada. Porém, começa-se a perceber seu potencial e linguagens de programação de cunho mais acadêmico, tais como Agda (NORELL, 2007) e Epigram (MCBRIDE, 2005), têm começado a utilizá-la. Entre as linguagens de programação de cunho mais comercial, o progresso é mais lento, mas extensões permitem simular algumas características de tipos dependentes na linguagem Haskell (MCBRIDE, 2002).

Para programadores, este trabalho se mostra relevante por permitir um contato inicial com a teoria de tipos dependentes em um contexto simplificado, fazendo com que seja mais fácil aprender suas peculiaridades. Além disso, utilizando o sistema desenvolvido neste trabalho, espera-se que o usuário possa verificar seu programa com regras de validação mais restritas que as encontradas na maioria das linguagens atuais, reduzindo o número de erros em tempo de execução.

Já para matemáticos, este trabalho é relevante pois irá permitir que escrevam seus teoremas de lógica de primeira ordem e os tenham validados pelo sistema de tipos da linguagem. No entanto, é importante ressaltar que o trabalho proposto não objetiva chegar à prova de teoremas automaticamente, apenas checar se estão corretos de acordo com os teoremas previamente definidos. Ainda que em alguns casos seja possível inferir as definições intermediárias necessárias para que um teorema seja provado automaticamente, isso está além dos objetivos do trabalho.

1.3 METODOLOGIA

O trabalho será desenvolvido observando as seguintes etapas:

a) levantamento bibliográfico: pesquisar material de referência sobre os assuntos abordados no trabalho, como provador interativo de teoremas, cálculo Lambda não-tipado e simplesmente tipado, sistemas de tipos e trabalhos correlatos;
b) elicitação dos requisitos: detalhar e reavaliar os requisitos, observando as necessidades levantadas durante a revisão bibliográfica;
c) definição da linguagem: definir a linguagem para descrever programas e teoremas de lógica de primeira ordem, usando definições regulares na especificação dos símbolos léxicos, gramática com a notação *Backus-Naur Form* (BNF) na especificação das regras sintáticas e esquemas de tradução na especificação da

semântica;

d) especificação do interpretador: especificar o interpretador com base na linguagem definida na etapa anterior. Serão utilizadas técnicas para modelagem de sistemas desenvolvidos com programação funcional;

e) implementação do interpretador: implementar o interpretador utilizando a linguagem Haskell e as seguintes ferramentas:
   - o compilador GHC na versão 7.8,
   - o programa Cabal para gerenciar o projeto,
   - o programa Haddock para gerar a documentação,
   - a biblioteca Parsec para desenvolver o analisador sintático;

f) testes: criar testes automatizados para validar as rotinas mais importantes do interpretador utilizando as bibliotecas HUnit e QuickCheck, além de realizar testes manuais para garantir o funcionamento do interpretador como um todo.

As etapas serão realizadas nos períodos relacionados no Quadro 1.

Quadro 1 - Cronograma

	2015									
	fev.		mar.		abr.		maio		jun.	
etapas / quinzenas	1	2	1	2	1	2	1	2	1	2
levantamento bibliográfico										
elicitação dos requisitos										
definição da linguagem										
especificação do interpretador										
implementação do interpretador										
testes										

2 REVISÃO BIBLIOGRÁFICA

Inicialmente, aborda-se o que são provadores interativos de teoremas, quais seus usos e limitações. Depois, apresenta-se o cálculo Lambda não-tipado, cuja compreensão é necessária para o entendimento dos assuntos seguintes. Então, faz-se uma breve introdução ao cálculo Lambda simplesmente tipado, que serve de base para os sistemas de tipos apresentados na sequência. Por fim, são abordados três trabalhos correlatos a este.

2.1 PROVADOR INTERATIVO DE TEOREMAS

Provadores interativos de teoremas, também conhecidos por assistentes de provas, são programas utilizados para auxiliar no desenvolvimento de provas formais. De maneira geral, eles permitem que um usuário interaja com um interpretador que valida as definições e os teoremas que o usuário escreve com uma linguagem formal.

Alguns assistentes de provas são mais complexos e automatizam determinados passos durante o desenvolvimento de uma prova. É o caso do sistema Coq (THE COQ DEVELOPMENT TEAM, 2014). Assim, o usuário não precisa digitar a demonstração de uma prova passo a passo, pois o sistema é capaz de inferir novas definições e teoremas a partir de dados que o usuário informou previamente.

Esses sistemas também podem ser utilizados para programação certificada (CHLIPALA, 2013). Nesse caso, escreve-se um programa cuja validade será verificada por um provador interativo de teoremas. Então, exporta-se esse programa para a linguagem de programação em que o restante do programa será escrito. Isso permite que o núcleo crítico de um programa seja certificado com regras de validação mais rígidas que as presentes na maioria das linguagens de programação. Um exemplo recente foi a verificação formal do microkernel seL4 (KLEIN et al., 2014).

2.2 CÁLCULO LAMBDA NÃO-TIPADO

O cálculo Lambda é um sistema formal para a representação de computações proposto por Church (1936), tendo grande importância na área da teoria da computabilidade. No ano seguinte, Turing (1937) demonstrou que a máquina de Turing e o cálculo Lambda são equivalentes em termos de computabilidade. Sendo assim, a escolha por um ou por outro se dá pelo problema em questão e pela preferência de quem pretende resolvê-lo. Uma analogia possível é dizer que a máquina de Turing está para linguagens de programação imperativas como o cálculo Lambda está para linguagens funcionais (TURNER, 2007).

De maneira informal, pode-se entender o cálculo Lambda como uma linguagem de

programação composta por apenas três tipos de termos (HUDAK, 2008): variáveis, abstrações (funções) e aplicações (aplicações de uma função a seu argumento). Assim, a função *identidade,* escrita na linguagem Java no Quadro 2, pode ser escrita na notação do cálculo Lambda como apresentado no Quadro 3.

Quadro 2 – A função *identidade* em Java

```
int identidade(int x) {
    return x;
}
```

Quadro 3 – A função *identidade* no cálculo Lambda

$\lambda x \,.\, x$

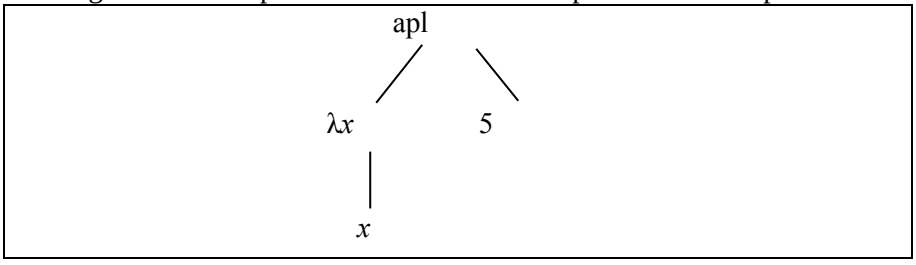
Já a aplicação de funções pode ser realizada concatenando-se o termo que define a função a ser computada e o termo que deve ser substituído. Para tornar as expressões mais legíveis e evitar erros de ambiguidade, podem-se utilizar parênteses. O Quadro 4 contém alguns exemplos de aplicações da função *identidade*, *dobro* e *soma dos quadrados*.

Quadro 4 – Exemplos de aplicações

$(\lambda x \,.\, x)\ 5$	resulta em 5
$(\lambda y \,.\, y + y)\ 7$	resulta em 14
$(\lambda(x, y) \,.\, x * x + y * y)\ (3, 4)$	resulta em 25

Outra interpretação para o cálculo Lambda é que ele representa a árvore sintática abstrata de uma expressão (JUNG, 2004). Assim, a expressão contida na primeira linha do Quadro 4 pode ser transformada na árvore representada na Figura 1 de forma trivial, em que "apl" significa a aplicação de uma abstração.

Figura 1 – Exemplo de árvore construída a partir de uma expressão



Essa relação explica porque o cálculo Lambda se tornou o arcabouço preferido dos autores de linguagens de programação funcionais, começando por McCarthy (1960), que especificou a linguagem LISP, e Landin (1966), que descreveu a linguagem ISWIM e influenciou o desenvolvimento de linguagens como Haskell, OCaml e Standard ML.

2.3 CÁLCULO LAMBDA SIMPLESMENTE TIPADO

A teoria dos tipos foi desenvolvida por Russell (1903) em resposta a algumas contradições encontradas por ele mesmo em sua teoria dos conjuntos. Da forma como foi elaborada, era possível descrever um conjunto R tal que $R = \{ w \mid w \notin w \}$, ou seja, R é o

conjunto que contém todos os conjuntos que não pertencem a eles mesmos. Porém, isso dá origem a um paradoxo, conhecido por Paradoxo de Russell: *R* é um conjunto que contém a si próprio se e somente se *R* não contiver a si próprio. A partir do uso de tipos, passa a ser possível distinguir entre objetos e predicados, predicados de predicados, entre outros, evitando-se assim esse paradoxo.

Mais tarde, Church (1940) simplificou essa teoria e a juntou com o cálculo Lambda, dando origem ao cálculo Lambda simplesmente tipado. Passam a existir dois tipos básicos, *i* (o tipo dos indivíduos) e *o* (o tipo das proposições), e o construtor de tipos →, definido por: sejam α e β tipos, então $\alpha \rightarrow \beta$, o tipo das funções de α para β, também é um tipo (COQUAND, 2014).

Dessa forma, tipos mais complexos podem ser construídos, como por exemplo:

a) $i \rightarrow i$: tipo das funções;

b) $i \rightarrow o$: tipo dos predicados;

c) $(i \rightarrow o) \rightarrow o$: tipo dos predicados de predicados.

Se *M* e *N* são termos do cálculo Lambda e α e β são tipos, então pode-se escrever $M : \alpha$ para expressar que *M* tem tipo α, dando origem às regras escritas no Quadro 5.

Quadro 5 – Regras do cálculo Lambda simplesmente tipado

$$\frac{N : \alpha \rightarrow \beta \qquad M : \alpha}{N\,M : \beta}$$

$$\frac{M : \beta \qquad [\, x : \alpha \,]}{\lambda x \,.\, M : \alpha \rightarrow \beta}$$

A primeira regra diz que se *N* é uma função de α para β e *M* tem tipo α, então a aplicação de *N* sobre *M* tem tipo β. A segunda significa que se *M* tem tipo β e *x* é definido para ser do tipo α, então a função definida por $\lambda x \,.\, M$ tem tipo $\alpha \rightarrow \beta$.

2.4 SISTEMAS DE TIPOS

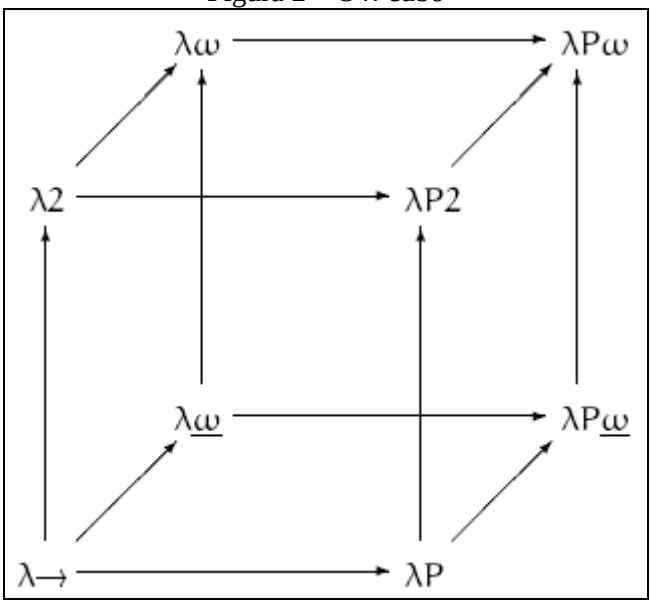
Intuitivamente, as regras do cálculo Lambda simplesmente tipado são aquelas que se esperam de uma linguagem de programação. Isso acontece porque os sistemas de tipos das linguagens de programação e a teoria de tipos estão intimamente relacionados. O objetivo de um sistema de tipos é associar tipos a várias construções, como valores, variáveis, expressões e funções, que depois são validados de acordo com as regras de teoria de tipos (PIERCE, 2002). Isso permite a redução no número de falhas em programas e melhores otimizações pelo compilador (CARDELLI, 2004).

Porém, em busca de uma maior expressividade, sistemas de tipos podem ser estendidos, dando-lhes novas características. Três dessas possíveis extensões ao sistema de

tipos do cálculo Lambda simplesmente tipado (λ→) são representadas no λ-cubo, concebido por Barendregt (1991) e ilustrado na Figura 2, que são: polimorfismo (λ2), operadores de tipos (λω) e tipos dependentes (λP), abordadas nas subseções seguintes.

Essas extensões são de interesse por serem ortogonais e poderem ser combinadas, dando origem a novos sistemas de tipos. Dentre as possíveis combinações, uma que recebe destaque é a que inclui as três extensões, sendo chamada de cálculo de construções (λPω).

Figura 2 – O λ-cubo



Fonte: Barendregt (1991, p. 126).

Outra extensão importante, mas que não faz parte do λ-cubo, são os subtipos, utilizados frequentemente em linguagens orientadas a objetos.

2.4.1 Polimorfismo

O polimorfismo (representado por λ2 no λ-cubo), também conhecido por sistema F ou cálculo Lambda de segunda ordem, permite a definição de funções de tipos para termos. Foi introduzido por Girard (1971) e posteriormente, de maneira independente, por Reynolds (1974). Assim como no cálculo Lambda simplesmente tipado se utiliza o símbolo λ para expressar uma função de termos para termos, no cálculo Lambda com polimorfismo se utiliza o símbolo Λ para expressar uma função de tipos para termos.

A primeira linha do Quadro 6 ilustra a definição da função *identidade* de forma tipada e polimórfica. De maneira sucinta, pode-se dizer que $\Lambda\alpha\ .\ \lambda x^{\alpha}\ .\ x$ é uma função de tipos para termos cujo termo resultante sempre terá tipo $\alpha \rightarrow \alpha$. Para exemplificar, na linha seguinte essa função de tipos para termos é aplicada ao tipo *int*, dando origem ao termo na terceira linha. Essa última expressão possui tipo *int* → *int*, já que se substituiu α por *int*.

Quadro 6 – A função *identidade* no cálculo Lambda com polimorfismo

$\Lambda\alpha \,.\, \lambda x^{\alpha} \,.\, x : \forall\alpha \,.\, \alpha \rightarrow \alpha$
$(\Lambda\alpha \,.\, \lambda x^{\alpha} \,.\, x : \forall\alpha \,.\, \alpha \rightarrow \alpha)\ int$
$\lambda x^{int} \,.\, x : int \rightarrow int$

Adaptando esse exemplo para a linguagem Java, tem-se o Quadro 7, onde uma função *identidade* é definida. Essa função não possui um tipo pré-definido, mas sabe-se que a função adotará o tipo sobre o qual for aplicada, e que esse será o mesmo tipo do parâmetro de entrada e do valor de retorno dessa função.

Quadro 7 – A função *identidade* com polimorfismo em Java

```
public static <A> A identidade(A x) {
    return x;
}
```

É importante ressaltar que o polimorfismo possui nomes diferentes para diferentes classes de programadores. Para programadores acostumados com linguagens orientadas a objetos, como é o caso do exemplo anterior, este assunto se refere a *generics* ou polimorfismo paramétrico. Por outro lado, programadores habituados a linguagens funcionais conhecem este conceito simplesmente por polimorfismo. Por exemplo, na linguagem Haskell, a função *identidade* pode ser escrita como no Quadro 8.

Quadro 8 – Exemplo de polimorfismo em Haskell

```
identidade :: a -> a
identidade x = x
```

2.4.2 Subtipos

Subtipos também dão origem a um tipo de polimorfismo, mas, devido a diferenças em como são tratados, são classificados à parte. Um subtipo é um tipo relacionado a outro tipo (o supertipo) por alguma noção de substituibilidade (PIERCE, 2002). Assim, seja τ um tipo e σ seu subtipo, uma função que possa ser aplicada sobre um termo de tipo τ também pode ser aplicada a um termo de tipo σ. Simbolicamente, isso é representado por $\sigma <: \tau$.

Novamente, é importante ressaltar que subtipos recebem nomes diferentes por diferentes classes de programadores. Programadores que utilizam linguagens funcionais conhecem este conceito por tipo soma. O Quadro 9 exemplifica o conceito mostrando como é implementado em Haskell: o tipo *Forma* possui dois subtipos, *Retangulo* e *Circulo*. Uma função que receba um argumento do tipo *Forma* pode receber também argumentos dos subtipos *Retangulo* e *Circulo*.

Quadro 9 – Exemplo de subtipos em Haskell

```
data Forma = Retangulo (Float, Float) (Float, Float)
           | Circulo (Float, Float) Float
```

Por outro lado, programadores acostumados com linguagens orientadas a objetos

chamam este conceito de polimorfismo, relacionado na maioria das linguagens a outro conhecido por herança. O Quadro 10 exemplifica a criação de três classes em Java, sendo que as classes *Retangulo* e *Circulo* são subclasses de *Forma.* Uma função que receba um argumento da classe *Forma* pode receber também argumentos das subclasses *Retangulo* e *Circulo.*

Quadro 10 – Exemplo de subtipos em Java

```
// Forma.java
class Forma {
}

// Retangulo.java
class Retangulo extends Forma {
    float x1, y1;
    float x2, y2;
}

// Circulo.java
class Circulo extends Forma {
    float x, y;
    float raio;
}
```

2.4.3 Operadores de tipos

A extensão de operadores de tipos (representados por $\lambda\underline{\omega}$ no λ-cubo) permite a definição de funções de tipos para tipos e trabalha com a noção de espécie (*kind*). Um sistema de espécies pode ser compreendido como um cálculo Lambda simplesmente tipado um nível acima, contendo um tipo primitivo representado por * e chamado de tipo, que é a espécie dos tipos de dados da linguagem (PIERCE, 2002). É a partir desse tipo primitivo e do operador → que se tem a origem de outras espécies:

a) *, chamado tipo, é a espécie à qual pertencem os tipos de dados "comuns", como inteiros, lógicos e caracteres;
b) * → *, é a espécie de tipos cujo construtor depende de outro tipo, como é o caso do tipo de uma lista, que pode ser uma lista de inteiros ou uma lista de lógicos;
c) * → * → *, é a espécie de tipos cujo construtor depende de dois outros tipos, como é o caso do tipo de um par, que pode ser qualquer combinação de dois tipos (um inteiro e um lógico, um caractere e uma palavra, dois inteiros, entre outros).

Deste modo, pode-se entender que espécies atuam como uma forma de metatipo. São poucas as linguagens que as utilizam, como Haskell e Scala, e mesmo nessas linguagens seu uso é limitado, servindo mais para indicar o número de argumentos que um construtor de tipo necessita.

2.4.4 Tipos dependentes

Os tipos dependentes (representados por λP no λ-cubo) permitem a definição de funções de termos para tipos. Em particular, isso permite a definição de funções cujo tipo de retorno depende do valor do argumento e não do tipo do argumento, como é no caso do polimorfismo, e de pares dependentes, em que o tipo do segundo elemento depende do primeiro (tipos-Π). Com pares dependentes passa a ser possível, por exemplo, definir pares em que o segundo elemento é obrigatoriamente maior que o primeiro (BOVE; DYBJER, 2009).

Embora as linguagens que utilizem tipos dependentes em sua totalidade sejam raras e pouco conhecidas, algumas linguagens mais conhecidas os utilizam em certos pontos do sistema de tipos. Um exemplo é a linguagem C++, a partir do padrão de 2011, que só permite a atribuição de um vetor a uma variável se ambos possuírem o mesmo tamanho, como exemplificado no Quadro 11.

Quadro 11 – Exemplo do uso de tipos dependentes na linguagem C++

```
std::array<int, 3> a1 = {1, 2, 3};
std::array<int, 3> a2 = {4, 5, 6};

a2 = a1;
```

Porém, em linguagens de programação em que tipos dependentes são utilizados mais extensivamente, é possível fazer com que o sistema de tipos realize verificações mais complexas. Por exemplo, no Quadro 12, define-se uma função *sort* que recebe uma lista qualquer e retorna uma lista ordenada. Embora isso possa ser realizado em qualquer linguagem, em Agda, o sistema de tipos garante que o algoritmo de ordenação realmente está correto e que a lista retornada está de fato ordenada.

Quadro 12 – Exemplo de uso de tipos dependentes em Agda

```
sort : List X → OList ⊥ ⊤
sort = foldr (λ x xs → insert x xs ⊥≤ ≤⊤) (nil ⊥≤)
```

Outro ponto de interesse dos tipos dependentes é sua relação com a lógica de primeira ordem. De acordo com o isomorfismo de Curry-Howard, existe uma equivalência entre lógica e sistemas de tipos, conforme apresentado no Quadro 13 (SØRENSEN; URZYCZYN, 2006). Para que um sistema de tipos possa tratar de fórmulas com quantificações universais ou existenciais, presentes na lógica de primeira ordem, é necessária a presença de tipos-Π e tipos-Σ, estudados dentro da teoria de tipos dependentes.

Quadro 13 – Equivalência entre lógica e sistemas de tipos

lógica	**sistemas de tipos**
quantificação universal	tipo-Π
quantificação existencial	tipo-Σ
consequência	tipo função
conjunção	tipo produto
disjunção	tipo soma
fórmula verdadeira	tipo unidade
fórmula falsa	tipo vazio

Fonte: adaptado de Chakraborty (2011).

2.5 TRABALHOS CORRELATOS

A seguir são apresentados três trabalhos com características semelhantes aos principais objetivos deste trabalho. O primeiro é uma linguagem de programação conhecida por Agda, o segundo é o assistente de provas Coq e o terceiro é um interpretador para uma linguagem com tipos dependentes.

2.5.1 Agda

A linguagem Agda foi desenvolvida por Norell (2007) em sua tese de doutorado como uma tentativa de aproximar a apresentação mais teórica e formal da teoria dos tipos de uma linguagem de programação de uso mais prático. Dois pontos interessantes da tese são:

a) a descrição de um sistema de módulos para a linguagem Agda, tornando-a mais próxima de uma linguagem completa, em vez de uma linguagem apenas para a demonstração dos conceitos estudados;
b) a apresentação da relação entre tipos dependentes e um provador de teoremas de lógica de primeiro grau.

2.5.2 Coq

Esse projeto desenvolvido pela The Coq Development Team (2014) é um sistema para gerenciamento de provas formais. Ele fornece uma linguagem formal para a escrita de definições matemáticas, algoritmos executáveis e teoremas junto com um ambiente semi-interativo para o desenvolvimento de provas checadas por computadores.

Em vez de utilizar exclusivamente a teoria de tipos dependentes, o Coq utiliza o cálculo de construções, uma variação do cálculo Lambda tipado com um nível de abstração maior que o cálculo Lambda com tipos dependentes. Baseando-se no λ-cubo, pode-se entender o cálculo de construções como um cálculo Lambda ao qual foram adicionados polimorfismo, operadores de tipos e tipos dependentes.

Dois destaques do Coq são sua capacidade de inferir provas através de métodos já

programados ou definidos pelo usuário, além de ser possível utilizá-lo para extrair programas verificados para outras linguagens, como Haskell, OCaml e Scheme.

2.5.3 Interpretador para uma linguagem com tipos dependentes

Nesse artigo, Löh, McBride e Swierstra (2010) apresentam uma introdução à implementação de sistemas de tipos com tipos dependentes, definindo e desenvolvendo uma linguagem funcional com tipos dependentes.

Embora se trate de uma introdução, o texto parte do pressuposto de que o leitor já possua algum conhecimento sobre teoria de tipos dependentes. Seu objetivo principal é preencher uma lacuna entre a bibliografia atual, mais formal e mais focada em teóricos da teoria dos tipos, e programadores interessados em linguagens funcionais.

Partindo da implementação de uma linguagem baseada no cálculo Lambda simplesmente tipado, os autores mostram que alterações são necessárias tanto na especificação como na implementação para que se chegue a uma linguagem com tipos dependentes. Além do artigo, os autores disponibilizam também o código fonte de um interpretador com o sistema de tipos e as regras de execução descritos no texto.

3 REQUISITOS DO SISTEMA A SER DESENVOLVIDO

O provador interativo de teoremas descrito nesta proposta deverá:

a) possuir uma linguagem que permita a escrita de programas e teoremas de lógica de primeira ordem (Requisito Funcional – RF);
b) disponibilizar um interpretador de linha de comando para a execução interativa dessa linguagem (RF);
c) permitir a execução dessa linguagem a partir da leitura de um arquivo texto (RF);
d) possuir um sistema de tipos capaz de validar os programas e os teoremas de lógica de primeira ordem (RF);
e) utilizar a teoria dos tipos dependentes para o desenvolvimento do sistema de tipos (Requisito Não Funcional – RNF);
f) ser implementado utilizando a linguagem Haskell e o compilador GHC versão 7.8 (RNF);
g) possuir testes automatizados implementados com as ferramentas HUnit e QuickCheck (RNF);
h) ser implementado na língua inglesa (RNF).

4 CONSIDERAÇÕES FINAIS

Como já mencionado, provadores interativos de teoremas reúnem os avanços em sistemas de tipos mais expressivos e no uso de ferramentas computacionais para auxílio no desenvolvimento de provas formais. Neste sentido, propõe-se desenvolver um assistente de provas baseado na teoria de tipos dependentes, uma área relativamente ainda pouco explorada dentro da teoria de tipos em que o tipo de um termo pode depender do valor de outro termo. Para isso, faz-se necessário especificar uma linguagem de programação e desenvolver um interpretador capaz de executá-la.

A respeito dos trabalhos correlatos apresentados, é possível afirmar que a linguagem Agda (NORELL, 2007) segue uma abordagem similar ao trabalho proposto, tanto no que tange a linguagem como o sistema de tipos desenvolvido para ela. Já o trabalho descrito por Löh, McBride e Swierstra (2010), embora possua objetivos parecidos, difere deste trabalho por ter um enfoque mais prático e não descrever em detalhes a teoria na qual a linguagem desenvolvida se baseia. Por fim, o sistema Coq (THE COQ DEVELOPMENT TEAM, 2014) é similar nos objetivos apresentados, mas vai além e permite a prova automática de teoremas. Além disso, o sistema Coq se distingue deste trabalho por se basear na teoria do cálculo das construções e não na teoria de tipos dependentes, embora ambos conceitos estejam relacionados (BARENDREGT, 1991).

Com o desenvolvimento deste provador interativo de teoremas, pretende-se permitir que programadores tenham um primeiro contato com a teoria de tipos dependentes, conhecendo suas peculiaridades, vantagens e desvantagens. Também pretende-se que matemáticos possam validar seus teoremas de lógica de primeira ordem de maneira interativa, utilizando o sistema de tipos da linguagem.

REFERÊNCIAS BIBLIOGRÁFICAS

APPEL, Kenneth; HAKEN, Wolfgang. Every planar map is four colorable - Part I: discharging. **Illinois Journal of Mathematics**, Ilinóis, v. 21, n. 3, p. 429-490, 1977. Disponível em: <http://projecteuclid.org/euclid.ijm/1256049011>. Acesso em: 31 ago. 2014.

AVIGAD, Jeremy; HARRISON, John. Formally verified mathematics. **Communications of the ACM**, New York, v. 57, n. 4, p. 66-75, Apr. 2014. Disponível em: <http://cacm.acm.org/magazines/2014/4/173219-formally-verified-mathematics/fulltext>. Acesso em: 31 ago. 2014.

BARENDREGT, Henk. Introduction to generalized type systems. **Journal of Functional Programming**, New York, v. 1, n. 2, p. 125-154, Apr. 1991. Disponível em: <http://repository.ubn.ru.nl/bitstream/handle/2066/17240/13256.pdf?sequence=1>. Acesso em: 7 set. 2014.

BOVE, Ana; DYBJER, Peter. Dependent types at work. In: BOVE, Ana et al. (Ed.). **Language engineering and rigorous software development**. Berlin: Springer-Verlag, 2009. p. 57-99. Disponível em: <http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf>. Acesso em: 7 set. 2014.

CARDELLI, Luca. Type systems. In: TUCKER, Allen B. (Ed.). **Computer science handbook**. 2nd ed. [S.l.]: Chapman & Hall/CRC, 2004. cap. 97. p. 1-41. Disponível em: <http://lucacardelli.name/papers/typesystems.pdf>. Acesso em: 7 set. 2014.

CHAKRABORTY, Subashis. **Curry-Howard-Lambek correspondence**. [S.l.]: [s.n.], 2011. Disponível em: <http://pages.cpsc.ucalgary.ca/~robin/class/617/projects-10/Subashis.pdf>. Acesso em: 7 set. 2014.

CHLIPALA, Adam. **Certified programming with dependent types**: a pragmatic introduction to the Coq proof assistant. Massachusetts: MIT Press, 2013. Disponível em: <http://adam.chlipala.net/cpdt/cpdt.pdf>. Acesso em: 8 set. 2014.

CHURCH, Alonzo. An unsolvable problem of elementary number theory. **American Journal of Mathematics**, Baltimore, v. 58, n. 2, p. 345-363, Apr. 1936. Disponível em: <http://phil415.pbworks.com/f/Church.pdf>. Acesso em: 6 set. 2014.

_____. A formulation of the simple theory of types. **The Journal of Symbolic Logic**, Cambridgeshire, v. 5, n. 2, p. 56-68, Jun. 1940. Disponível em: <http://www.classes.cs.uchicago.edu/archive/2007/spring/32001-1/papers/church-1940.pdf>. Acesso em: 7 set. 2014.

COQUAND, Thierry. Type theory. In: ZALTA, Edward N. (Ed.). **The Stanford encyclopedia of philosophy**. Stanford: Stanford University, 2014. Disponível em: <http://plato.stanford.edu/entries/type-theory/>. Acesso em: 7 set. 2014.

GIRARD, Jean-Yves. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des Coupures dans l'analyse et la théorie des types. In: SCANDINAVIAN LOGIC SYMPOSIUM, 2nd, 1970, Oslo. **Proceedings...** Amsterdam: North-Holland Pub. Co., 1971. p. 63-92. Disponível em: <http://bookzz.org/book/538944/8434c2>. Acesso em: 7 set. 2014.

HUDAK, Paul; JONES, Mark. **Haskell vs. Ada vs. C++ vs. Awk vs. …**: an experiment in software prototyping productivity. New Haven: Department of Computer Science, Yale University, 1994. Disponível em: <http://haskell.cs.yale.edu/?post_type=publication&p=366>. Acesso em: 31 ago. 2014.

HUDAK, Paul. **A brief and informal introduction to the Lambda Calculus**. [New Haven], 2008. Disponível em: <http://www.cs.yale.edu/homes/hudak/CS201S08/lambda.pdf>. Acesso em 6 set. 2014.

JUNG, Achim. **A short introduction to the Lambda Calculus**. [Birmingham], 2004. Disponível em: <http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf>. Acesso em: 6 set. 2014.

KLEIN, Gerwin et al. Comprehensive formal verification of an OS microkernel. **ACM Transactions On Computer Systems**. New York, v. 32, n. 1, p. 1-70, Feb. 2014. Disponível em: <http://www.nicta.com.au/pub?doc=7371>. Acesso em: 8 set. 2014.

LANDIN, Peter J. The next 700 programming languages. **Communications of the ACM**, New York, v. 9, n. 3, p. 157-166, Mar. 1966. Disponível em: <http://www.thecorememory.com/Next_700.pdf>. Acesso em: 7 set. 2014.

LÖH, Andres; MCBRIDE, Conor; SWIERSTRA, Wouter. A tutorial implementation of a dependently typed Lambda Calculus. **Fundamenta Informaticae**, Amsterdam, v. 102, n. 2, p. 177-207, 2010. Disponível em: <http://www.andres-loeh.de/LambdaPi/LambdaPi.pdf>. Acesso em: 24 ago. 2014.

MCBRIDE, Conor. Faking it: simulating dependent types in Haskell. **Journal of Functional Programming**, New York, v. 12, n. 5, p. 375-392, 2002. Disponível em: <http://strictlypositive.org/faking.ps.gz>. Acesso em: 31 ago. 2014.

MCBRIDE, Conor; MCKINNA, James. The view from the left. **Journal of Functional Programming**, New York, v. 14, n. 1, p. 69-111, 2004. Disponível em: <http://strictlypositive.org/view.ps.gz>. Acesso em: 31 ago. 2014.

MCBRIDE, Conor. Epigram: practical programming with dependent types. In: INTERNATIONAL CONFERENCE ON ADVANCED FUNCTIONAL PROGRAMMING, 5th, 2005, Tartu. **Proceedings…** Berlim: Springer, 2005. p. 130-170. Disponível em: <http://strictlypositive.org/epigram-notes.pdf>. Acesso em: 31 ago. 2014.

MCCARTHY, John. Recursive functions of symbolic expressions and their computation by machine. **Communications of the ACM**, New York, v. 3, n. 4, p. 184-195, Apr. 1960. Disponível em: <http://www-formal.stanford.edu/jmc/recursive.pdf>. Acesso em: 7 set. 2014.

NORELL, Ulf. **Towards a practical programming language based on dependent type theory**. 2007. 166 f. Thesis (Doctor of Philosophy) - Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Göteborg. Disponível em: <https://synrc.com/publications/cat/Functional%20Languages/Agda/PracticalDependent.pdf>. Acesso em: 31 ago. 2014.

PIERCE, Benjamin C. **Types and programming languages**. Massachusetts: MIT Press, 2002.

REYNOLDS, John C. Towards a theory of type structure. In: COLLOQUE SUR LA PROGRAMMATION, 1974, Paris. **Proceedings…** London: Springer-Verlag, 1974. p. 408-423. Disponível em: <http://www.cse.chalmers.se/edu/year/2010/course/DAT140_Types/Reynolds_theotypestr.pdf>. Acesso em: 7 set. 2014.

RUSSELL, Bertrand. **The principles of mathematics**. 2nd ed. [S.l]: W. W. Norton & Company, 1903. Disponível em: <http://fair-use.org/bertrand-russell/the-principles-of-mathematics/index>. Acesso em: 7 set. 2014.

SØRENSEN, Morten H.; URZYCZYN, Pawel. **Lectures on the Curry-Howard isomorphism**. [S.l.]: Elsevier, 2006. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.7385&rep=rep1&type=pdf>. Acesso em: 7 set. 2014.

THE COQ DEVELOPMENT TEAM. **The Coq proof assistant**. [S.l.], 2014. Disponível em: <http://coq.inria.fr/>. Acesso em: 31 ago. 2014.

TURING, Alan M. Computability and λ-definability. **The Journal of Symbolic Logic**, Cambridgeshire, v. 2, n. 4, p. 153-163, Dec. 1937. Disponível em: <http://cs.simons-rock.edu/cmpt312/turing.pdf>. Acesso em: 6 set. 2014.

TURNER, David. Church's thesis and functional programming. In: OLSZEWSKI, Adam; WOLENSKI, Jan; JANUSZ, Robert (Ed.). **Church's thesis after 70 years**. Berlin: Ontos Verlag, 2007. p. 518-543. Disponível em: <http://www.cs.kent.ac.uk/people/staff/dat/miranda/ctfp.pdf>. Acesso em: 6 set. 2014.