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
2015

2015/1-03

ANDRÉ RAMACIOTTI DA SILVA

PROVADOR INTERATIVO DE TEOREMAS COM TIPOS DEPENDENTES

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Profª. Joyce Martins, Mestre - Orientadora

BLUMENAU
2015

2015/1-03

PROVADOR INTERATIVO DE TEOREMAS COM TIPOS DEPENDENTES

Por

ANDRÉ RAMACIOTTI DA SILVA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente:	Profª. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Profª. Gabriele Jennrich Bambineti, Esp. – FURB
Membro:	Prof. Matheus Carvalho Viana, Dr. – FURB

Blumenau, 8 de julho de 2015

Dedico este trabalho a todos que estudaram, estudam ou estudarão teoria de tipos e linguagens de programação.

AGRADECIMENTOS

A meus pais, por todo o apoio e educação.

Aos meus amigos, pelo apoio e pelas risadas.

À minha orientadora, Joyce Martins, pelas revisões e pelas sugestões de como deixar este trabalho melhor.

A todos os autores presentes nas referências, cujas obras e ferramentas foram fundamentais para o desenvolvimento deste trabalho.

Às comunidades do *Hacker News, Lambda the Ultimate* e *r/programming*, por me manterem atualizado e sem os quais eu talvez não tivesse conhecido e me interessado pelo assunto deste trabalho.

Yeah... That went well.

Capt. Malcolm Reynolds

RESUMO

Esta monografia de trabalho de conclusão de curso apresenta a especificação e a implementação de um provador interativo de teoremas baseado em uma linguagem formal com tipos dependentes. Com isso, mostra-se uma forma em que computadores podem auxiliar no desenvolvimento matemático e se introduz a teoria dos tipos dependentes a programadores em uma linguagem simplificada. De forma mais detalhada, objetiva-se especificar uma linguagem para descrever programas e teoremas de lógica de primeira ordem e um sistema de tipos baseado na teoria de tipos dependentes, implementar um interpretador de linha de comando e validar os programas e os teoremas de lógica de primeira ordem escritos nessa linguagem. Para isso, durante o desenvolvimento, é elaborada uma definição formal para a linguagem especificada através da notação *Backus-Naur Form* (BNF) e implementado um interpretador de linha de comando com a linguagem Haskell e a biblioteca Parsec. Ao final, todos os objetivos são alcançados, embora o uso da teoria de tipos dependentes faça com que a definição de certos tipos e funções se tornem mais complexa, além de restringir os teoremas que podem ser provados àqueles com provas construtivas. Faz-se também uma comparação com as linguagens Agda, Coq e a desenvolvida por Löh, McBride e Swierstra (2010), mostrando que a abordagem de Coq de separar as provas das definições de funções torna o código mais legível.

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

ABSTRACT

This document presents the specification and implementation of an interactive theorem prover based on a formal language with dependent types. With it, a way in which computers may help in the mathematical development is shown, and the theory of dependent types is introduced to programmers in a simplified language. In more details, the objectives are to specify a language for describing programs and first-order logic theorems and a type system based on the theory of dependent types, to implement a command line interpreter, and to validate programs and first-order logic theorems written in this language. For this, during the development, a formal definition for this language is developed through the Backus-Naur Form (BNF) notation, and a command line interpreter is implemented with the Haskell language and the Parsec library. Finally, all the objectives are reached, although the use of dependent type theory makes the definition of certain types and functions more complex, besides restricting the theorems that can be proved to those with constructive proofs. A comparison is made between the Agda and Coq language, and the language developed by Löh, McBride and Swierstra (2010), showing that Coq's approach of separating proofs from function definitions makes code more readable.

Key-words: Proof assistant. Programming languages. Type systems. Dependent type theory.

LISTA DE FIGURAS

Figura 1 – Exemplo de uma árvore construída a partir de uma expressão 17
Figura 2 – O λ-cubo 20
Figura 3 – Ambiente de desenvolvimento integrado do Coq 28
Figura 4 – Diagrama de caso de uso 32
Figura 5 – Diagrama de classes 36

LISTA DE QUADROS

Quadro 1 – A função `identidade` em Java 17
Quadro 2 – A função `identidade` no cálculo Lambda 17
Quadro 3 – Exemplo de aplicações 17
Quadro 4 – Regras do cálculo Lambda simplesmente tipado 18
Quadro 5 – Exemplo de *currying* de uma função com três argumentos 19
Quadro 6 – A função `identidade` no cálculo Lambda com polimorfismo 21
Quadro 7 – A função `identidade` com polimorfismo em Java 21
Quadro 8 – Exemplo de polimorfismo em Haskell 21
Quadro 9 – Exemplo de subtipos em Haskell 22
Quadro 10 – Exemplo de subtipos em Java 22
Quadro 11 – Exemplo do uso de tipos dependentes na linguagem C++ 23
Quadro 12 – Exemplo de tipo polimórfico em Java 23
Quadro 13 – Equivalência entre lógica de predicados e sistemas de tipos 24
Quadro 14 – Exemplo de função `sort` em Agda 24
Quadro 15 – Trechos do módulo `Sort` 25
Quadro 16 – Sessão interativa com a linguagem Agda 26
Quadro 17 – Trechos do módulo `Coq.Sorting.Mergesort` 27
Quadro 18 – Trechos das definições padrões do interpretador de Löh, McBride e Swierstra 29
Quadro 19 - Definição do tipo `Bool` 29
Quadro 20 - Definição da computação de um valor de tipo `Bool` 30
Quadro 21 – Requisitos funcionais 31
Quadro 22 – Requisitos não funcionais 31
Quadro 23 – Detalhamento do caso de uso `Executar interpretador` 32
Quadro 24 – Especificação da linguagem 33
Quadro 25 – Exemplos de definições de tipos 34
Quadro 26 – Exemplos de definições de funções 34
Quadro 27 – Função `parseConstructor` 38
Quadro 28 – Equivalência entre BNF e Haskell (função `parseConstructor`) 38
Quadro 29 – Função `parseType` 39
Quadro 30 - Equivalência entre BNF e Haskell (função `parseType`) 39
Quadro 31 – Funções `evalProgram` e `evalToplevel` 40

Quadro 32 – Funções `isValidType` e `isTypeAssignable` 41
Quadro 33 – Função `tryEvalExpression` 41
Quadro 34 - Função `bindTypeVars` 42
Quadro 35 - Início de uma sessão interativa 42
Quadro 36 - Entrada de um comando longo 43
Quadro 37 - Uso do comando de impressão 43
Quadro 38 - Erros de execução 44
Quadro 39 - Arquivo `exemplo.dt` 44
Quadro 40 - Execução do arquivo `exemplo.dt` 44
Quadro 41 – Função `pairAdd` 45
Quadro 42 – Prova que a soma de dois números ímpares é par 45
Quadro 43 – Comparativo entre os trabalhos correlatos e o protótipo desenvolvido 46
Quadro 44 – Trechos do módulo `Sort` em Agda 47
Quadro 45 – Trechos do módulo `Coq.Sorting.Mergesort` em Coq 47
Quadro 46 – Função de ordenação com tipos dependentes 48
Quadro 47 – Definições de tipos e constantes 55
Quadro 48 – Definições de funções com argumentos 56
Quadro 49 – Desconstrução de argumentos 57
Quadro 50 – Utilização de tipos dependentes 57
Quadro 51 – Função de ordenação com tipos dependentes 59

LISTA DE ABREVIATURAS E SIGLAS

BNF – *Backus-Naur Form*

FFI – *Foreign Function Interface*

GHC – *Glasgow Haskell Compiler*

RF – Requisito Funcional

RNF – Requisito Não Funcional

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS 15
1.2 ESTRUTURA 15
2 FUNDAMENTAÇÃO TEÓRICA 16
2.1 PROVADOR INTERATIVO DE TEOREMAS 16
2.2 CÁLCULO LAMBDA NÃO-TIPADO 17
2.3 CÁLCULO LAMBDA SIMPLESMENTE TIPADO 18
2.4 SISTEMAS DE TIPOS 19
2.4.1 Polimorfismo 20
2.4.2 Subtipos 21
2.4.3 Operadores de tipos 22
2.4.4 Tipos dependentes 23
2.5 TRABALHOS CORRELATOS 24
2.5.1 Agda 25
2.5.2 Coq 26
2.5.3 Interpretador para uma linguagem com tipos dependentes 28
3 DESENVOLVIMENTO 31
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 31
3.2 ESPECIFICAÇÃO 31
3.2.1 Caso de uso 32
3.2.2 Especificação da gramática 32
3.2.3 Diagrama de tipos 35
3.3 IMPLEMENTAÇÃO 37
3.3.1 Técnicas e ferramentas utilizadas 37
3.3.2 Desenvolvimento do protótipo 38
3.3.2.1 Implementação do analisador sintático 38
3.3.2.2 Implementação do analisador semântico 39
3.3.3 Operacionalidade da implementação 42
3.4 RESULTADOS E DISCUSSÕES 44
4 CONCLUSÕES 49
4.1 EXTENSÕES 50

REFERÊNCIAS 51
APÊNDICE A – Exemplo de código escrito na linguagem do protótipo..... 55

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.

Na computação, Hudak e Jones (1994) afirmam que 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.

Dentre as possíveis fundamentações teóricas para o desenvolvimento de provadores interativos de teoremas, 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).

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.

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, um programador pode utilizar-se do sistema

[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.

de tipos desenvolvido neste trabalho para 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 permite 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 desenvolvido 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.

1.1 OBJETIVOS

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) implementar um interpretador de linha de comando para essa linguagem;
d) validar os programas e os teoremas de lógica de primeira ordem escritos nessa linguagem.

1.2 ESTRUTURA

O presente trabalho está organizado em quatro capítulos, sendo o primeiro esta introdução. O segundo capítulo apresenta os aspectos teóricos necessários para o entendimento do trabalho e aborda tópicos como provadores interativos de teoremas, cálculo Lambda não-tipado e simplesmente tipado e sistemas de tipos. Também são descritos alguns trabalhos correlatos.

O terceiro capítulo refere-se ao desenvolvimento deste trabalho, detalhando os requisitos do protótipo implementado, sua especificação e implementação. São apresentados também os resultados obtidos ao término do trabalho.

Por último, o quarto capítulo traz as conclusões obtidas através deste trabalho e aborda aspectos que ficaram em aberto, servindo como sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Inicialmente, neste capítulo, 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. Em seguida, 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.

Em vez de se basearem na teoria dos conjuntos, como é comum em outras divisões da matemática, os provadores interativos de teoremas utilizam a teoria de tipos. Segundo Asperti (2009), isso facilita a detecção de erros sintáticos e semânticos durante a análise das provas escritas pelo usuário. A presença de tipos também torna mais fácil automatizar certos passos durante o desenvolvimento de uma prova.

Além desses pontos, o uso da teoria de tipos pode ser justificado por suas relações com a lógica construtiva. Como descreve Fernandes (2009), formulações como a interpretação de Brouwer-Heyting-Kolmogorov e a correspondência de Curry-Howard permitem traçar equivalências entre as computações realizadas na teoria de tipos e a lógica construtiva.

Conforme escreve Chlipala (2013), esses sistemas também podem ser utilizados para programação certificada. Nesse caso, escreve-se um programa cuja validade será verificada por um provador interativo de teoremas. Em seguida, 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 por 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 1, pode ser escrita na notação do cálculo Lambda como apresentado no Quadro 2.

Quadro 1 – A função `identidade` em Java

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

Quadro 2 – A função `identidade` no cálculo Lambda

```
λx . x
```

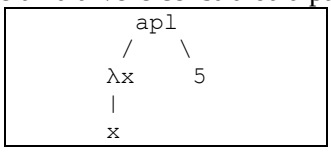
Já a aplicação de funções pode ser realizada concatenando-se o termo que define a função a ser computada e o termo que deve ser substituído. Para tornar as expressões mais legíveis e evitar erros de ambiguidade, podem-se utilizar parênteses. O Quadro 3 contém alguns exemplos de aplicações da função `identidade`, `dobro` e `soma dos quadrados`.

Quadro 3 – Exemplo de aplicações

```
(λx . x) 5                          resulta em 5
(λy . y + y) 7                      resulta em 14
(λ(x, y) . x * x + y * y) (3, 4)    resulta em 25
```

Como mostra Jung (2004), outra interpretação para o cálculo Lambda é que ele representa a árvore sintática abstrata de uma expressão. Assim, a expressão contida na primeira linha do Quadro 3 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 uma 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 $\rightarrow$, 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 4. 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 $\alpha \rightarrow \beta$.

Quadro 4 – Regras do cálculo Lambda simplesmente tipado

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

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

Em algumas situações, é necessário escrever funções que recebam mais de um argumento. Isso pode ser feito através de n-uplas, como no último exemplo do Quadro 3. Outra técnica comumente utilizada é chamada de *currying*, em homenagem a Haskell Curry (BARENDREGT; BARENDSEN, 2000). Através de seu uso, pode-se entender uma função

de dois argumentos como tendo tipo $\alpha \rightarrow (\beta \rightarrow \gamma)$ em vez de $(\alpha \times \beta) \rightarrow \gamma$. Desta forma, uma função que receba como argumento uma 2-upla cujos elementos têm tipos α e β, pode ser reescrita como uma função que receba um argumento de tipo α e retorne outra função de tipo $\beta \rightarrow \gamma$.

O Quadro 5 ilustra como uma função que recebe três argumentos através de uma 3-upla pode ser reescrita como uma cadeia de funções que recebem apenas um argumento através do uso de *currying*. Esse quadro também demonstra como a aplicação dessas funções difere.

Quadro 5 – Exemplo de *currying* de uma função com três argumentos

```
(λ(a, b, c) . a * b + c)                 : (α × β × γ) → δ
(λ(a, b, c) . a * b + c) (5, 8, 2)
(5 * 8 + 2)
42                                       : δ


(λa . (λb . (λc . a * b + c)))           : α → (β → (γ → δ))
(λa . (λb . (λc . a * b + c))) 5
(λb . (λc . 5 * b + c))                  : β → (γ → δ)
(λb . (λc . 5 * b + c)) 8
(λc . 5 * 8 + c)                         : γ → δ
(λc . 5 * 8 + c) 2
(5 * 8 + 2)
42                                       : δ
```

No primeiro exemplo, nas linhas agrupadas na parte superior do quadro, tem-se uma função que recebe três argumentos de tipos α, β e γ e que retorna um valor de tipo δ. Na forma como foi escrita, ela recebe todos os parâmetros em uma única 3-upla, *(5, 8, 2)*, e retorna o valor computado, *42*. Em seguida, no grupo de linhas na parte inferior do quadro, reescreve-se essa função com o uso de *currying*. Nesse segundo exemplo, a aplicação da função é feita de forma parcial, começando-se pelo primeiro argumento, *5*. Isso resulta em uma função intermediária (demarcada pela anotação de tipo $\beta \rightarrow (\gamma \rightarrow \delta))$ que recebe outro argumento e retorna mais uma função. Novamente, faz-se uma aplicação parcial, agora com o segundo argumento, *8*, e obtém-se mais uma função intermediária (demarcada por $\gamma \rightarrow \delta))$. Finalmente, aplica-se o último argumento, *2*, e chega-se ao mesmo valor computado, *42*.

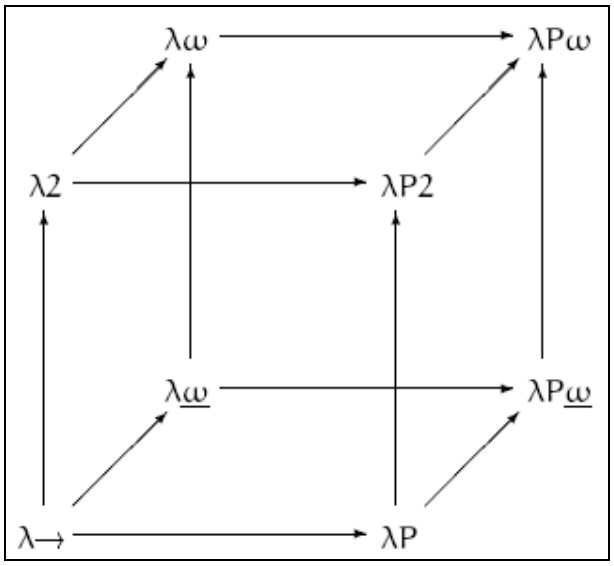
2.4 SISTEMAS DE TIPOS

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.

Figura 2 – O λ-cubo



Fonte: Barendregt (1991, p. 126).

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). 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

```
Λα . λx^α . x : ∀α . α → α
(Λα . λx^α . x : ∀α . α → α) int
λx^int . x : int → 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;
}
```

O'Sullivan, Stewart, e Goerzen (2008) escrevem que esse tipo de polimorfismo possui nomes diferentes para diferentes classes de programadores. No contexto de Java e C#, ele geralmente é chamado de genéricos. Já em linguagens como Haskell, ele é chamado de polimorfismo paramétrico. 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 $\sigma <: \tau$.

Assim como acontece com polimorfismo, subtipos também recebem diferentes nomes por diferentes classes de programadores. Programadores que utilizam linguagens funcionais conhecem este conceito por tipos algébricos de dados (O'SULLIVAN; STEWART; GOERZEN, 2008). 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, embora não sejam o mesmo (COOK; HILL; CANNING, 1990). 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 $\rightarrow$ 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) $* \rightarrow *$, 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) $* \rightarrow * \rightarrow *$, 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

Segundo Bove e Dybjer (2009), os tipos dependentes (representados por λP no λ-cubo) são tipos que dependem de elementos de outros tipos. Um exemplo comum são as linguagens que permitem que o tamanho de um vetor seja especificado como parte de seu tipo. O Quadro 11 ilustra a definição de dois vetores, `a1` e `a2`, na linguagem C++. Ambos vetores são do tipo `std::array<int, 3>`, fazendo com que a atribuição da última linha seja válida. Caso qualquer um desses vetores fosse de tipo diferente (`std::array<int, 4>`, por exemplo), essa atribuição seria inválida e haveria um erro de compilação.

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;
```

Ainda segundo Bove e Dybjer (2009), embora certas construções sejam possíveis tanto pelo uso de polimorfismo como pelo uso de tipos dependentes, a interpretação é diferente. Por exemplo, no Quadro 12 é definida uma lista de inteiros na linguagem Java, que utiliza polimorfismo. Do ponto de vista dessa teoria, isso é possível porque `Integer` é um tipo. Se Java utilizasse a teoria dos tipos dependentes, isso seria possível porque `Integer` seria um elemento do tipo `Tipo`, que pode ser entendido como um conjunto que contém todos os tipos definidos no programa, assim como `3` é um elemento do tipo `int` no Quadro 11. Resumindo, no polimorfismo, tipos podem depender somente de outros tipos, enquanto na teoria dos tipos dependentes, tipos podem depender de elementos de outras formas, incluindo outros tipos.

Quadro 12 – Exemplo de tipo polimórfico em Java

```
List<Integer> inteiros;
```

Conforme escrevem Sørensen e Urzyczyn (2006), o desenvolvimento da teoria dos tipos dependentes ocorre em conjunto com o do isomorfismo de Curry-Howard, segundo o qual existe uma equivalência entre a lógica de predicados construtiva e os tipos de um sistema de tipos com tipos dependentes, conforme mostra o Quadro 13. Com base nesse isomorfismo, Martin-Löf (1985) desenvolve um sistema de tipos que recebe seu nome. Embora a intenção inicial de Martin-Löf fosse elaborar uma fundação para a matemática construtiva, seu sistema de tipos também serviu de base para a elaboração do assistente de provas Coq (THE COQ DEVELOPMENT TEAM, 2014) e da linguagem Agda (NORELL, 2007).

De acordo com essa equivalência, quando se tem uma função *f* de tipo $\alpha \rightarrow \beta$, e ela é aplicada sobre um valor *a* de tipo α, chega-se a um valor do tipo β. Essa construção é equivalente ao *modus ponens* quando se tem $P \rightarrow Q$ e P, e conclui-se que Q. Assim, um sistema de tipos com tipos dependentes possui construções equivalentes à quantificação universal ($\forall x$), quantificação existencial ($\exists x$), consequência ou implicação ($\rightarrow$), conjunção ($\wedge$), disjunção ($\vee$) e aos valores de verdadeiro e falso.

Quadro 13 – Equivalência entre lógica de predicados e sistemas de tipos

lógica de predicados	**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).

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 que em outras linguagens. Alguns exemplos são a definição de um par *(a, b)* em que *a* é obrigatoriamente menor que *b* ou de uma função que soma os elementos de duas listas, par a par, desde que ambas possuam exatamente o mesmo tamanho. Para ilustrar, no Quadro 14 mostra-se uma função `sort`, elaborada por Mazzoli (2013), que recebe uma lista qualquer (`List`) e retorna uma lista ordenada (`OList`). 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 14 – Exemplo de função `sort` em Agda

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

Fonte: adaptado de Mazzoli (2013).

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 (NORELL, 2007), o segundo é o assistente de provas Coq (THE COQ DEVELOPMENT TEAM, 2014) e o terceiro é um interpretador para uma linguagem com tipos dependentes (LÖH; MCBRIDE; SWIERSTRA, 2010).

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.

Diferentemente do Coq, apresentado na próxima subseção, que permite a separação entre definições de funções e de teoremas a serem provados, programas escritos em Agda precisam adicionar as propriedades que serão provadas nas próprias definições dos tipos e das funções. No Quadro 15 está um exemplo de como uma função `isort` pode ser escrita de forma a garantir que a lista resultante está de fato ordenada (MAZZOLI, 2013).

Quadro 15 – Trechos do módulo `Sort`

```
data ⊥XT : Set where
  T ⊥ : ⊥XT
  ⟦_⟧ : X → ⊥XT

data _≤^_ : Rel ⊥XT where
  ⊥≤^     : ∀ {x} → ⊥ ≤^ x
  ≤^T     : ∀ {x} → x ≤^ T
  ≤-lift  : ∀ {x y} → x ≤ y → ⟦ x ⟧ ≤^ ⟦ y ⟧

data OList (l u : ⊥XT) : Set where
  nil  : l ≤^ u → OList l u
  cons : ∀ x (xs : OList ⟦ x ⟧ u) → l ≤^ ⟦ x ⟧ → OList l u

insert : ∀ {l u} x → OList l u → l ≤^ ⟦ x ⟧ → ⟦ x ⟧ ≤^ u → OList l u
  insert y (nil _)         l≤y y≤u = cons y (nil y≤u) l≤y
  insert y (cons x xs l≤x) l≤y y≤u with y ≤? x
  insert y (cons x xs l≤x) l≤y y≤u | left  y≤x = cons y
    (cons x xs (≤-lift y≤x)) l≤y
  insert y (cons x xs l≤x) l≤y y≤u | right y>x =
    cons x (insert y xs ([ ≤-lift , (λ y≤x → absurd (y>x y≤x)) ]
    (total x y)) y≤u) l≤x

isort : List X → OList ⊥ T
  isort = foldr (λ x xs → insert x xs ⊥≤^ ≤^T) (nil ⊥≤^)
```

Fonte: adaptado de Mazzoli (2013).

Esse código define uma função `isort`, que faz a ordenação de uma lista através de repetidas aplicações da função `insert`. Para que as validações desejadas pudessem ser aplicadas, foi necessário definir um novo tipo, `OList`, e outros tipos auxiliares, `≤^` e `⊥XT`. Isso faz com que o código escrito em Agda difira significativamente de código escrito em outras linguagens funcionais sem tipos dependentes.

Desde a publicação da tese, foram adicionadas novas características à linguagem. Uma que merece destaque é a adição de uma *Foreign Function Interface* (FFI) para a execução de código escrito em Haskell a partir de Agda e vice versa (DEVRIESE, 2014). Isso permite que o núcleo do programa seja escrito com as verificações adicionais que tipos dependentes promovem, mas partes auxiliares do programa podem ser escritas em outra linguagem quando o uso de tipos dependentes se mostrar incômodo.

Apesar de seu objetivo ser uma linguagem de programação e não um provador interativo de teoremas, existe um modo interativo para desenvolvimento com a linguagem Agda. Ele facilita a definição de funções, indicando quais são os casos para os quais ela ainda não foi definida, conforme mostra o Quadro 16. Nele, demonstra-se uma sessão interativa em que dois trechos de código escritos como `{ }0` e `{ }1` estão destacados, por terem sido detectados pelo interpretador da linguagem como pedaços em que o código está incompleto.

Quadro 16 – Sessão interativa com a linguagem Agda

```
module demo1A where

data Bool : Set where
  true : Bool
  false : Bool

not : Bool -> Bool -- '->' is input with '\to'
not true = { }0
not false = { }1
```

Fonte: adaptado de Maydwell (2012).

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 (BARENDREGT, 1991).

Dois destaques do Coq são sua capacidade de inferir provas por meio 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, permitindo a interação entre código escrito em Coq e nessas linguagens.

Graças à sua capacidade de inferir provas e seu objetivo de ser um provador interativo de teoremas em vez de apenas uma linguagem de programação, propriedades sobre tipos e funções podem ser escritas de forma mais sucinta. O Quadro 17 ilustra essa característica do Coq através de trechos de código do módulo `Coq.Sorting.Mergesort` (THE COQ DEVELOPMENT TEAM, 2015b). Isso distingue Coq dos demais trabalhos correlatos, em que teoremas e demais propriedades devem embutidas nas definições de funções e tipos de alguma forma.

Quadro 17 – Trechos do módulo `Coq.Sorting.Mergesort`

```
Fixpoint merge_list_to_stack stack l :=
  match stack with
  | [] => [Some l]
  | None :: stack´ => Some l :: stack´
  | Some l´ :: stack´ => None :: merge_list_to_stack stack´ (merge l´ l)
  end.

Fixpoint merge_stack stack :=
  match stack with
  | [] => []
  | None :: stack´ => merge_stack stack´
  | Some l :: stack´ => merge l (merge_stack stack´)
  end.

Fixpoint iter_merge stack l :=
  match l with
  | [] => merge_stack stack
  | a::l´ => iter_merge (merge_list_to_stack stack [a]) l´
  end.

Definition sort := iter_merge [].

Fixpoint SortedStack stack :=
  match stack with
  | [] => True
  | None :: stack´ => SortedStack stack´
  | Some l :: stack´ => Sorted l /\ SortedStack stack´
  end.

Theorem Sorted_merge_list_to_stack : forall stack l,
  SortedStack stack -> Sorted l -> SortedStack (merge_list_to_stack stack l).
```

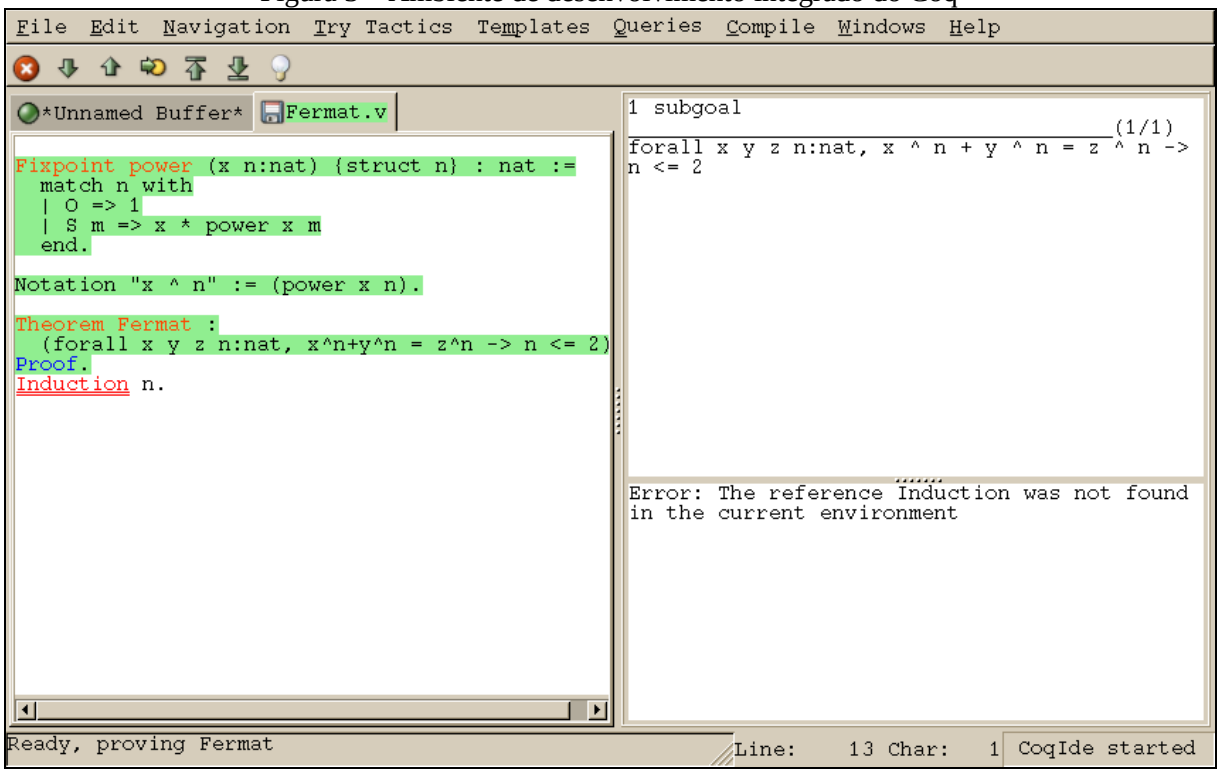
Fonte: adaptado de The Coq Development Team (2015b).

Esse código define uma função chamada `sort`, que é implementada através das funções recursivas `iter_merge`, `merge_stack` e `merge_list_to_stack`. Em sequência, no mesmo módulo, funções auxiliares e teoremas são definidos, provando que a implementação de `sort` está correta. Exemplos desse tipo de verificação são a função auxiliar `SortedStack` e o teorema `Sorted_merge_list_to_stack`. Essa separação entre a definição das funções e das propriedades a serem provadas permite que o código seja escrito de forma mais similar à como seria escrito em uma linguagem funcional sem tipos dependentes.

O Coq possui um ambiente de desenvolvimento integrado, oferecendo uma interface gráfica para a escrita de funções, teoremas, propriedades e demais trechos de código. A Figura

3 apresenta uma sessão de uso desse ambiente para uma tentativa de se provar o teorema de Fermat através do Coq.

Figura 3 – Ambiente de desenvolvimento integrado do Coq



Fonte: The Coq Development Team (2015a).

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 (LÖH, 2009a).

Devido ao enfoque ser maior à fundamentação teórica e à implementação que à usabilidade da linguagem final, programas escritos na linguagem desenvolvida nesse artigo

tendem a ser de difícil compreensão. Por exemplo, no Quadro 18 se demonstra o código necessário para se concatenar dois vetores.

Quadro 18 – Trechos das definições padrões do interpretador de Löh, McBride e Swierstra

```
let append =
( \ a -> vecElim a
         (\ m _ -> forall (n :: Nat) . Vec a n -> Vec a (plus m n))
         (\ _ v -> v)
         (\ m v vs rec n w -> Cons a (plus m n) v (rec n w)))
::  forall (a :: *) (m :: Nat) (v :: Vec a m) (n :: Nat) (w :: Vec a n).
    Vec a (plus m n)
```

Fonte: adaptado de Löh (2009b).

Além disso, novos tipos devem ser implementados diretamente no interpretador, em Haskell, ou devem ser definidos com base em algum tipo já pré-definido. Por exemplo, no Quadro 19 demonstra-se a definição do tipo `Bool`, realizada sobre o tipo `Fin 2`, e a de valores `False` e `True`, realizada com base nos números naturais que já estavam definidos. É necessário também definir a forma como valores desse tipo são computados (`boolElim`), tomando como base funções que já estavam definidas, como se mostra no Quadro 20.

Quadro 19 - Definição do tipo `Bool`

```
-- type of booleans
let Bool = Fin 2
-- constructors
let False = FZero 1
let True  = FSucc 1 (FZero 0)
```

Fonte: adaptado de Löh (2009b).

Devido a essas dificuldades no uso da linguagem desenvolvida para esse interpretador, optou-se por mantê-lo como trabalho correlato, por sua relevância teórica, mas preferiu-se não implementar uma função de ordenação para compará-lo aos outros trabalhos. A comparação será feita, assim, através das outras características apresentadas pelo trabalho.

Dado seu escopo reduzido, algumas das características dos outros trabalhos correlatos não estão presentes nesse interpretador. Em particular, ele não conta com uma interface semi-interativa e um sistema de módulos, como Agda e Coq possuem, e também não permite executar código escrito em outras linguagens, como Agda permite com Haskell, e Coq com as linguagens para as quais ele extrai programas certificados.

Quadro 20 - Definição da computação de um valor de tipo `Bool`

```
-- eliminator
let boolElim = ( \ m mf mt ->
finElim
( nat2Elim (\ n -> Fin n -> *)
           (\ _ -> Unit) (\ _ -> Unit)
           (\ x -> m x)
           (\ _ _ _ -> Unit) )
( nat1Elim ( \ n -> nat1Elim (\ n -> Fin (Succ n) -> *)
                             (\ _ -> Unit)
                             (\ x -> m x)
                             (\ _ _ _ -> Unit)
                             n (FZero n))
           U mf (\ _ _ -> U) )
( \ n f _ -> finElim
             ( \ n f -> nat1Elim (\ n -> Fin (Succ n) -> *)
                                 (\ _ -> Unit)
                                 (\ x -> m x)
                                 (\ _ _ _ -> Unit)
                                 n (FSucc n f) )
             ( natElim ( \ n -> natElim (\ n -> Fin (Succ (Succ n)) -> *)
                                        (\ x -> m x)
                                        (\ _ _ _ -> Unit)
                                        n (FSucc (Succ n) (FZero n)) )
                       mt (\ _ _ -> U) )
             ( \ n f _ -> finElim
                          (\ n f -> natElim (\ n -> Fin (Succ (Succ n)) -> *)
                                            (\ x -> m x)
                                            (\ _ _ _ -> Unit)
                                            n (FSucc (Succ n) (FSucc n f)))
                          (\ _ -> U)
                          (\ _ _ _ -> U)
                          n f )
             n f )
2 )
:: forall (m :: Bool -> *) . m False -> m True -> forall (b :: Bool) . m b
```

Fonte: adaptado de Löh (2009b).

3 DESENVOLVIMENTO

As seções a seguir descrevem os requisitos, a especificação, a implementação e a operacionalidade do protótipo, abordando sucintamente as ferramentas utilizadas nas respectivas etapas do processo. Ao fim, são mostrados os resultados obtidos com este trabalho.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O provador interativo de teoremas é um interpretador de linha de comando para uma linguagem formal em que é possível a escrita de programas e teoremas de lógica de primeira ordem. Além da interação direta com o usuário, esse interpretador deve também ser capaz de processar os programas escritos nessa linguagem a partir da leitura de um arquivo texto.

A seguir são mostrados os Requisitos Funcionais (RF) e Não Funcionais (RNF) atendidos pelo protótipo, apresentados respectivamente no Quadro 21 e no Quadro 22.

Quadro 21 – Requisitos funcionais

Requisitos funcionais (RF)
RF01: o provador interativo de teoremas deverá possuir uma linguagem que permita a escrita de programas e teoremas de lógica de primeira ordem.
RF02: o provador interativo de teoremas deverá permitir a execução interativa de programas escritos nessa linguagem.
RF03: o provador interativo de teoremas deverá permitir a execução de programas escritos nessa linguagem a partir da leitura de um arquivo texto.
RF04: o provador interativo de teoremas deverá possuir um sistema de tipos capaz de validar os programas e os teoremas de lógica de primeira ordem.

Quadro 22 – Requisitos não funcionais

Requisitos não funcionais (RNF)
RNF01: o provador interativo de teoremas deverá utilizar a teoria de tipos dependentes para o desenvolvimento do sistema de tipos.
RNF02: o provador interativo de teoremas deverá ser implementado com a linguagem Haskell e o compilador *Glasgow Haskell Compiler* (GHC).
RNF03: o provador interativo de teoremas deverá ter testes automatizados implementados com a ferramenta HUnit.
RNF04: o provador interativo de teoremas deverá ser implementado na língua inglesa.

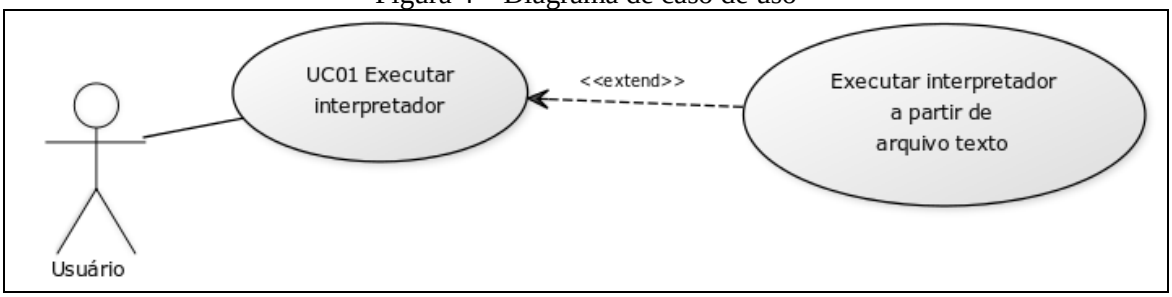
3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação do protótipo. Inicialmente, é detalhado o caso de uso com o diagrama próprio para este fim da *Unified Modeling Language* (UML). Em seguida, utiliza-se a notação *Backus-Naur Form* (BNF) para especificar a gramática da linguagem usada no protótipo. Por fim, os tipos e suas relações são descritos através de um diagrama de classes também da UML.

3.2.1 Caso de uso

O protótipo implementado atenderá a apenas um caso de uso (UC), `Executar interpretador`, com uma possível extensão, `Executar interpretador a partir de arquivo texto`, conforme mostrado na Figura 4.

Figura 4 – Diagrama de caso de uso



Esse caso de uso especificado no Quadro 23, descreve uma sessão de uso do protótipo. No cenário principal, o usuário interage com interpretador através de uma linha de comando, que executa cada comando digitado. No fluxo alternativo 01, descreve-se a utilização do interpretador com a extensão `Executar interpretador a partir de arquivo texto`.

Quadro 23 – Detalhamento do caso de uso `Executar interpretador`

UC01 – Executar interpretador: permite a interação através do interpretador	
Pré-condições	Nenhuma.
Cenário principal	01) O usuário inicia a execução do interpretador. 02) O interpretador aguarda a entrada de uma linha de comando. 03) O usuário digita a linha de comando e pressiona a tecla *Enter*. 04) O interpretador executa o comando digitado. 05) Retorna para o passo 02.
Fluxo alternativo 01	No passo 01, caso o usuário informe um arquivo de texto para ser executado: 01.01) O interpretador executa os comandos presentes no arquivo. 01.02) O interpretador encerra a execução.
Fluxo alternativo 02	No passo 04 (ou 01.01 no fluxo alternativo 01), caso o usuário tenha digitado um comando de impressão (instrução `print`): 04.01) O interpretador mostra o resultado da execução da linha de comando.
Fluxo alternativo 03	No passo 04, caso o usuário digite o comando para sair (instrução `exit`): 04.01) O interpretador encerra a execução.
Exceção 01	No passo 04 (ou 01.01 no fluxo alternativo 01), caso o usuário tenha digitado uma linha de comando inválida, é exibida uma mensagem informando o erro detectado.
Pós-condições	Nenhuma.

3.2.2 Especificação da gramática

A linguagem do protótipo foi desenvolvida com base na especificação do Quadro 24, escrita na notação BNF. Exemplos de código escritos nessa linguagem podem ser encontrados no Apêndice A.

Quadro 24 – Especificação da linguagem

```
<program>       ::= ε | <program-opt> <program>
<program-opt>   ::= <comment> | <type> | <function> | <print>

<comment>       ::= {- <text> -}

<type>          ::= type <type-id> : <signature> <const-opt> .
<const-opt>     ::= ε | where <const-list>
<const-list>    ::= <const> | <const> ; <const-list>
<const>         ::= <id> <arg-opt> : <signature> <cond-opt>
<cond-opt>      ::= ε | | <type-exp>

<function>      ::= func <func-id> <func-def> .
<func-id>       ::= <id> : <signature> | <id> : <signature> ; <func-id>
<func-def>      ::= where <func-list>
<func-list>     ::= <func-impl> | <func-impl> ; <func-list>
<func-impl>     ::= <id> <arg-opt> = <expression>

<signature>     ::= <sig-type > | <sig-type> -> <signature>
<sig-type>      ::= <type-id> | ( <type-exp> )

<arg-opt>       ::= ε | <arg-list>
<arg-list>      ::= <arg> | <arg> " " <arg-list>
<arg>           ::= <id> | ( <expression> ) | ( <type-exp> )

<print>         ::= print <print-list> .
<print-list>    ::= <print-arg> | <print-arg> ; <print-list>
<print-arg>     ::= <any-id> | ( <expression> )

<expression>    ::= <any-id> <exp-aux>
<type-exp>      ::= <type-id> <exp-aux>

<exp-aux>       ::= ε | <exp-list>
<exp-list>      ::= <expression> | <expression> " " <exp-list>

<any-id>        ::= <id> | <type-id>
<type-id>       ::= <uppercase-char> <id-opt>
<id>            ::= <lowercase-char> <id-opt>
<id-opt>        ::= ε | <lowercase-char> <id-opt> | <uppercase-char> <id-opt>
```

Conforme especificam as regras definidas a partir de `<program>` e `<program-opt>`, um programa é representado por zero ou mais construções que podem ser formadas a partir de `<comment>`, `<type>`, `<function>` ou `<print>`. Qualquer texto aparecendo entre os caracteres **`{-`** e **`-}`** é um comentário (`<comment>`) e será ignorado pelo interpretador.

Em seguida, `<type>` especifica como se dá a definição de tipos. Utiliza-se a palavra-chave `type`, seguida por um identificador de tipo (`<type-id>`), o símbolo dois-pontos, uma assinatura (`<signature>`), a definição de construtores (`<const-opt>`) e o símbolo ponto. A especificação de construtores para um tipo (`<const-opt>`) é opcional, podendo ser formada pela palavra-chave `where` seguida por uma lista de construtores (`<const-list>`). Essa lista é formada por pelo menos um construtor (`<const>`), sendo que as definições de construtores são separadas entre si pelo símbolo ponto-e-vírgula. Cada construtor é definido por um identificador (`<id>`), uma lista de argumentos (`<arg-opt>`), o símbolo dois-pontos, uma assinatura (`<signature>`) e uma restrição (`<cond-opt>`) opcional. Caso se opte por utilizar

uma restrição, ela é dada pelo símbolo barra vertical e por uma expressão de tipo (<type-exp>). O Quadro 25 ilustra uma definição de tipo sem construtores (Void) e uma com dois construtores (LessOrEqual), sendo que a segunda possui uma restrição.

Quadro 25 – Exemplos de definições de tipos

```
type Void : Type.

type LessOrEqual : Nat -> Nat -> Type where
  lessZero zero    y       : Nat -> Nat -> (LessOrEqual zero y);
  lessSuc  (suc x) (suc y) : Nat -> Nat -> (LessOrEqual (suc x) (suc y))
    | LessOrEqual x y.
```

A definição de funções (<function>) é similar à definição de tipos. Começa-se com a palavra-chave func, seguida por seus identificadores (<func-id>) e sua definição (<func-def>), finalizando com o símbolo ponto. Diferentemente da definição de tipos, que é feita de forma individual, é possível definir mais de uma função ao mesmo tempo, necessário para os casos de funções mutuamente recursivas. Assim, os identificadores que seguem a palavra-chave func (<func-id>) são compostos por um identificador (<id>), o símbolo dois-pontos e uma assinatura (<signature>). Quando se estiver definindo funções mutuamente recursivas, cada par identificador-assinatura é separado pelo símbolo ponto-e-vírgula. Por sua vez, a definição de função (<func-def>) é dada pela palavra-chave where, seguida por uma lista de implementações (<func-list>) separadas umas das outras pelo símbolo ponto-e-vírgula. Cada implementação (<func-impl>) é dada pelo identificador da função (<id>), seus argumentos (<arg-opt>), o símbolo igual e uma expressão que será executada quando a função for chamada (<expression>). O Quadro 26 mostra a definição de uma função cuja única implementação não possui argumentos (six) e de duas funções mutuamente recursivas (isEven e isOdd).

Quadro 26 – Exemplos de definições de funções

```
func six : Nat where
  six = suc (suc (suc (suc (suc (suc zero))))).

func isOdd : Nat -> Bool; isEven : Nat -> Bool where
  isOdd  zero    = false;
  isEven zero    = true;
  isOdd  (suc x) = isEven x;
  isEven (suc x) = isOdd x.
```

Comum às duas formas de definições está o conceito de assinatura (<signature>). Uma assinatura é uma lista composta por pelo menos um identificador de tipo (<type-id>) ou uma expressão de tipo (<type-exp>), sendo que as várias ocorrências são separadas pelo símbolo ->. A assinatura representa o tipo de outro tipo, um construtor ou uma função, e trabalha com o conceito de *currying*. Por exemplo, uma função que verifique se um número é

par tem assinatura (ou tipo) `Nat -> Bool`, e uma função que retorne o maior de três números possui assinatura (ou tipo) `Nat -> Nat -> Nat -> Nat`.

Outra regra utilizada tanto na definição de tipos como de funções é a definição de argumentos (`<arg-opt>`), que é opcional, podendo ser uma lista (`<arg-list>`), formada por um ou mais argumentos (`<arg>`) separados por espaços. Esses argumentos, por sua vez, podem ser um identificador (`<id>`), uma expressão (`<expression>`) ou uma expressão de tipo (`<type-exp>`).

A última regra que pode ser utilizada na formação de um programa são os comandos de impressão (`<print>`). Quando se deseja que o interpretador imprima alguma coisa, utiliza-se a palavra-chave `print` seguida por uma lista de expressões (`<print-list>`), sendo que as expressões são separadas entre si pelo símbolo ponto-e-vírgula. Essas expressões (`<print-arg>`) podem ser formadas por um único identificador (`<any-id>`) ou por expressão composta entre parênteses.

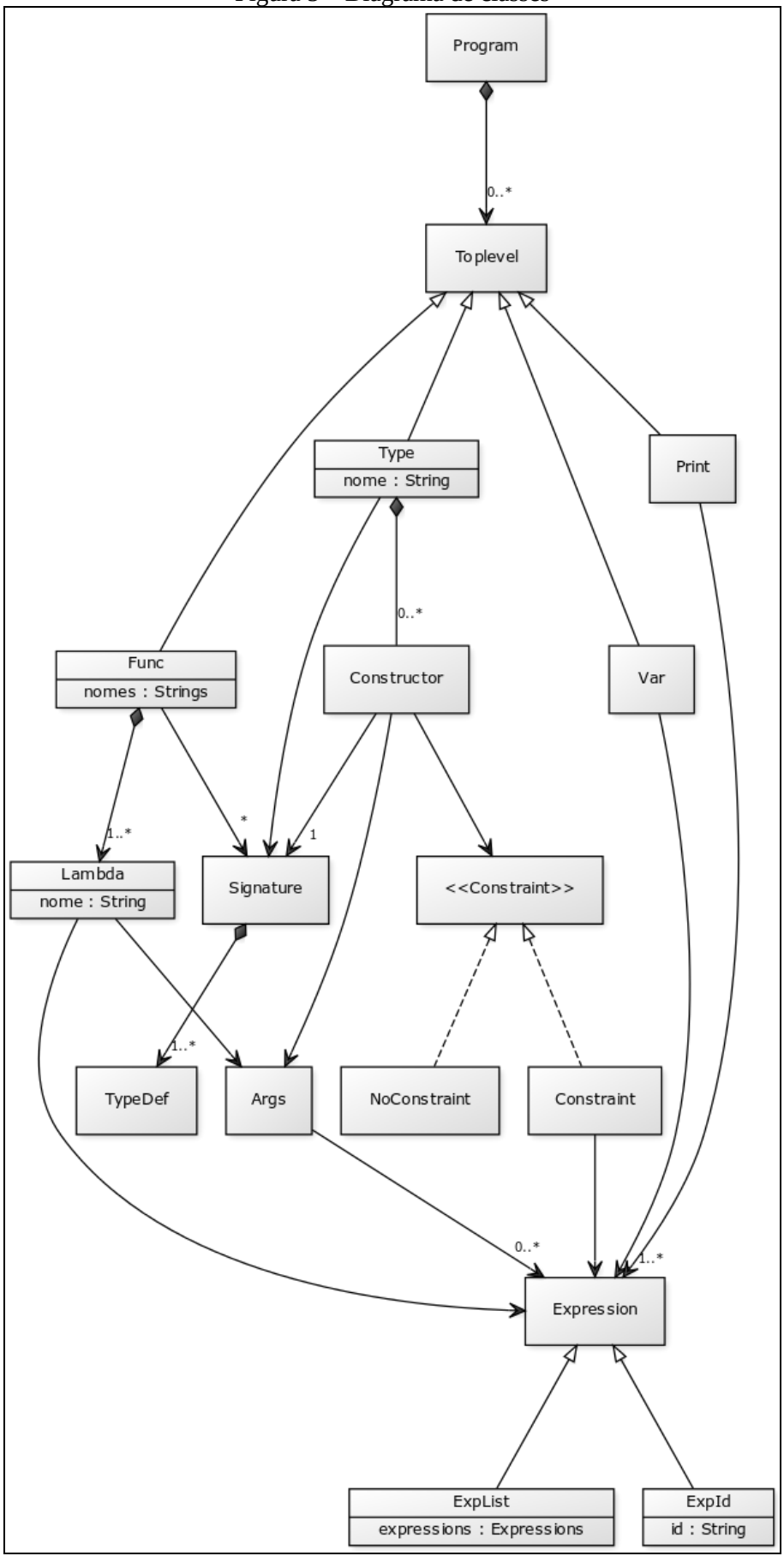
Além de expressões (`<expression>`), existe uma regra especial de expressões para quando se espera obrigatoriamente por um tipo (`<type-exp>`), que difere das expressões convencionais por seu primeiro (e possivelmente único) identificador ser um identificador de tipo (`<type-id>`) e não um identificador qualquer (`<any-id>`). Um identificador de tipo (`<type-id>`) tem uma letra maiúscula como primeiro caracter, enquanto um identificador convencional (`<id>`) tem uma letra minúscula como primeiro caracter.

3.2.3 Diagrama de tipos

O diagrama de classes da UML ilustra a estrutura e a relação entre as classes de um projeto. Apesar de Haskell não possuir classes e métodos no mesmo sentido que linguagens orientadas a objetos possuem, é possível utilizar esse diagrama para representar os tipos e suas relações.

Os tipos definidos se assemelham à gramática da linguagem, como mostra a Figura 5. O tipo principal é o programa (`Program`), que contém uma coleção de construções de nível superior (`Toplevel`), subdivididas em definições de tipo (`Type`), função (`Func`), além de comandos de impressão (`Print`) e de variáveis (`Var`). Desses, o único tipo que não possui uma relação direta com a especificação da linguagem é `Var`, que é usado internamente no interpretador para armazenar o valor de variáveis durante a execução de uma função.

Figura 5 – Diagrama de classes



Por sua vez, o tipo `Type` é formado por seu nome, uma assinatura (`Signature`) e uma coleção de construtores (`Constructor`). O tipo `Constructor` também é similar à sua representação na gramática da linguagem e possui seu nome, argumentos (`Args`), assinatura (`Signature`) e uma restrição (`Constraint`), que pode ser nula (`NoConstraint`) ou conter uma expressão (`Expression`).

De forma análoga, o tipo `Func` possui uma coleção de pares formados por um nome e uma assinatura (`Signature`). Diferentemente do tipo `Type`, que possui apenas um nome e uma assinatura, `Func` necessita de uma lista para os casos em que se estão definindo funções mutualmente recursivas. Esse tipo também conta com uma lista de `Lambda`, que serve dois propósitos: representar as implementações de funções mutuamente recursivas e armazenar implementações alternativas para uma mesma função. Esse segundo cenário aparece quando se utiliza o casamento de padrões, característica descrita no Apêndice A.

Compartilhados entre esses dois tipos estão a assinatura (`Signature`), representada por uma coleção de `TypeDef`, e os argumentos (`Args`), formados por uma lista de expressões (`Expression`).

Entre os tipos de nível superior, `Print` e `Var` são mais simples, sendo compostos, respectivamente, por uma lista de expressões (`Expression`) e uma única expressão. Essas expressões, que também são utilizadas por outros tipos, podem ser um identificador (`ExpId`) ou uma expressão composta por outras expressões (`ExpList`).

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas para a implementação do protótipo. Ao final desta seção é exibida também a operacionalidade da implementação, ilustrando uma sessão de uso típica do provador interativo de teoremas.

3.3.1 Técnicas e ferramentas utilizadas

O protótipo foi implementado com a linguagem Haskell e o compilador GHC (MARLOW, 2015), seguindo o paradigma da programação funcional. Para a codificação foi utilizado o editor de textos Emacs, devido a sua boa integração com as ferramentas de desenvolvimento do Haskell.

Para o desenvolvimento do analisador sintático, foi utilizada a biblioteca Parsec, uma biblioteca de código aberto desenvolvida especificamente para esse fim. Como mostram O'Sullivan, Stewart, e Goerzen (2008), um de seus principais atrativos é concisão, sendo

possível escrever analisadores sintáticos para linguagens relativamente complexas em poucas linhas de código.

Com o fim de facilitar o processo de desenvolvimento, foram utilizadas também as ferramentas: Cabal (THE CABAL DEVELOPMENT TEAM, 2015), responsável por gerenciar o projeto em Haskell e suas dependências; Haddock (MARLOW, 2014), para realizar a documentação do código fonte; e HUnit (HERINGTON, 2013), para automatizar o processo de testes.

3.3.2 Desenvolvimento do protótipo

O desenvolvimento do protótipo foi dividido em duas etapas: na primeira foi implementado o analisador sintático da linguagem e na segunda, o sistema de tipos. A seguir, são apresentados os trechos de código mais significativos dessas etapas.

3.3.2.1 Implementação do analisador sintático

Os analisadores léxico e sintático foram unificados em um único módulo do programa, `DependentTypes.Parser`. Para seu desenvolvimento foi utilizada a biblioteca Parsec, que permite a descrição desses tipos de analisadores em uma notação próxima à utilizada na especificação da gramática. Por exemplo, o Quadro 27 mostra a função responsável por fazer a análise de um construtor de um tipo, equivalente às regras do não-terminal `<const>` na gramática da linguagem. Pode-se fazer uma comparação entre como a regra foi escrita na notação BNF, `<const> ::= <id> <arg-opt> : <signature> <cond-opt>`, e no código em Haskell, chegando-se às equivalências do Quadro 28.

Quadro 27 – Função `parseConstructor`

```
parseConstructor :: Parser Constructor
parseConstructor = do
  constructorId <- many letter
  args <- parseArgs
  char ':'
  spaces
  signature <- parseSignature
  constraint <- parseConstraint
  return $ Constructor constructorId args signature constraint
```

Quadro 28 – Equivalência entre BNF e Haskell (função `parseConstructor`)

regra BNF	trecho do código
`<const> ::=`	`parseConstructor = do`
`<id>`	`constructorId <- many letter`
`<arg-opt>`	`args <- parseArgs`
`:`	`char ':'`
`<signature>`	`signature <- parseSignature`
`<cond-opt>`	`constraint <- parseConstraint`

Outras funções apresentam equivalências entre o não-terminal e as respectivas regras na notação BNF e o código em Haskell de forma similar. Outro exemplo pode ser visto na

função `parseType`, responsável por analisar um `<type>`, dado por `<type> ::= type <type-id> : <signature> <const-opt>`. O código da função é apresentado no Quadro 29 e a equivalência com a regra na notação BNF é apresentada no Quadro 30. Algumas funções adicionais são necessárias para que se ignorem caracteres de espaço, mas em geral as duas ficam similares.

Quadro 29 – Função `parseType`

```
parseType :: Parser Toplevel
parseType = do
  string "type"
  spaces
  typeId <- many letter
  spaces
  char ':'
  spaces
  signature <- parseSignature
  constructors <- parseConstructors
  return $ Type typeId signature constructors
```

Quadro 30 - Equivalência entre BNF e Haskell (função `parseType`)

regra BNF	trecho de código
`<type> ::=`	`parseType = do`
`type`	`string "type"`
`<type-id>`	`typeId <- many letter`
`:`	`char ':'`
`<signature>`	`signature <- parseSignature`
`<constructors>`	`constructors <- parseConstructors`

Outras partes do interpretador usam o módulo `DependentTypes.Parser` por meio da função `parser`, que recebe o programa como uma `String` e retorna um valor do tipo `Either ParseError Program`. Assim, ao se chamar essa função, o resultado pode ser um erro na análise gramatical, representado por um valor do tipo `ParseError`, definido pela biblioteca Parsec, ou um valor do tipo `Program`, explicado na seção em que se mostra o diagrama de tipos (seção 3.2.3).

3.3.2.2 Implementação do analisador semântico

O analisador semântico, em conjunto com o sistema de tipos, é implementado no módulo `DependentTypes.Semantic`. Suas principais funções utilizadas pelo restante do interpretador são `nullEnv`, responsável por criar uma tabela de variáveis vazia, e `evalProgram`, responsável por executar um programa.

Como mostra o Quadro 31, a função `evalProgram` recebe três parâmetros: uma ação, uma tabela de variáveis e um programa. A ação é um comando que será executado quando algum valor tiver de ser impresso. Embora o padrão seja imprimir esses valores em tela, a possibilidade de utilizar uma ação alternativa facilitou o desenvolvimento de algumas partes dos testes automatizados. A tabela de variáveis é a mesma retornada pela função `nullEnv`.

Conforme o programa é executado, ela deixa de ser uma tabela vazia e passa a ter associações entre nomes e tipos ou funções. Por fim, o programa é um valor do tipo `Program` conforme ele é retornado da função `parse` do módulo `DependentTypes.Parser`.

Quadro 31 – Funções `evalProgram` e `evalToplevel`

```
-- | Evaluates a gramatically valid program.
evalProgram :: (String -> IO ()) -> Env -> Program -> IO ()
evalProgram action env (Program ts) = forM_ ts $ evalToplevel action env

-- | Evaluates a toplevel construct.
evalToplevel :: (String -> IO ()) -> Env -> Toplevel -> IO ()
evalToplevel action env t@(Type name sig cons) = do
  checkTypeSignature env name sig
  modifyIORef env (name `Map.insert` t)
  forM_ cons $ \c@(Constructor consName _ sig _) -> do
                      checkConsSignature env consName sig
                      modifyIORef env (consName `Map.insert` t)
evalToplevel action env f@(Func ss lambdas) = do
  checkFuncSignature env ss
  forM_ ss $ \s@(name, _) -> modifyIORef env (name `Map.insert` Func [s] lambdas)
  checkFuncLambdas env ss lambdas
evalToplevel action env print@(Print exp) = do
  e <- readIORef env
  evalExp <- forM exp $ evalExpression e
  action $ showExpressions evalExp
```

A execução de um programa é, na verdade, a execução de cada uma de suas construções de nível superior. Assim, na prática, a função `evalProgram` apenas delega a execução dessas construções para a função `evalToplevel`, que utiliza casamento de padrões para executar trechos de código diferentes dependendo se a construção sendo executada é a definição de um tipo, de uma função ou um comando de impressão.

Os casos para as definições seguem um formato similar, tanto para tipos como para funções. Primeiro verifica-se a validade das definições em funções como `checkTypeSignature` e `checkFuncSignature` e, então, atualiza-se a tabela de variáveis, com base na função `modifyIORef`, adicionando as definições realizadas.

Embora sejam funções diferentes, a maioria das verificações chegam à função `isTypeAssignable`, que recebe a tabela de variáveis, uma lista de expressões e uma lista de expressões de tipo, como mostra o Quadro 32. Ela é responsável por verificar se as expressões passadas como argumento casam com as expressões de tipo.

Quando a construção sendo executada é um comando de impressão, chama-se a função `tryEvalExpression`. Como mostra o Quadro 33, faz-se uma verificação adicional durante a execução, por meio da função `checkInferredTypes`, e então executam-se as expressões de forma recursiva. Por exemplo, ao se executar a expressão `add (suc six) seven`, primeiro se executa `six`. Depois, com o resultado da execução anterior, executa-se `suc six`. Então, executa-se `seven`. Por fim, executa-se a função `add` com os resultados obtidos.

Quadro 32 – Funções `isValidType` e `isTypeAssignable`

```
-- | Checks if a type name refers to an existing type with the same arity.
isValidType :: Map String Toplevel -> TypeDef -> Bool
isValidType m (TypeId name) =
  case name `Map.lookup` m of
    Just (Type n (Signature [x]) _) -> n == name
    _                               -> isAsciiLower (head name)
isValidType m (DepType name es) =
  case name `Map.lookup` m of
    Just (Type n (Signature ts) _) -> n == name && isDepTypeAssignable m es ts
    _                              -> isAsciiLower (head name)

-- | Checks if a list of expressions is assignable to a list of types.
isTypeAssignable :: Map String Toplevel -> [Expression] -> [TypeDef] -> Bool
isTypeAssignable m [] [] = False
isTypeAssignable m [ExpId expId] [TypeId typeId] =
  case expId `Map.lookup` m of
    Just (Type typeName _ cons)         -> typeName == typeId || isAsciiLower
(head typeId)
    Just (Func [(_, (Signature ss))] _) -> last ss == TypeId typeId ||
isAsciiLower (head typeId)
    Just (Var _)                        -> True
    Nothing                             -> True
isTypeAssignable m [ExpId expId] [DepType typeId _] =
  case expId `Map.lookup` m of
    Just (Type typeName _ cons)         -> typeName == typeId
    Just (Func [(_, (Signature ss))] _) -> isValidSignature (last ss)
    Just (Var _)                        -> True
    Nothing                             -> True
  where
    isValidSignature (DepType depType _) = depType == typeId
    isValidSignature (TypeId typeName)   = typeName == typeId
isTypeAssignable m (expHead@(ExpId expId):expTail) ts =
  1 + length expTail == length ts &&
  isTypeAssignable m [expHead] [(last ts)] &&
  (all (==True) $ zipWith isTypeAssignable' expTail ts)
  where
    isTypeAssignable' exp@(ExpId expId) t = isTypeAssignable m [exp] [t]
    isTypeAssignable' (ExpList (expHead:expTail)) t = isTypeAssignable m
[expHead] [t]
```

Quadro 33 – Função `tryEvalExpression`

```
-- | Tries to evaluate an expression.
tryEvalExpression :: Map String Toplevel -> Expression -> Either String
Expression
tryEvalExpression e exp@(ExpId expId) = do
  newEnv <- checkInferredTypes e [exp]
  newEnv <- return (e `Map.union` newEnv)
  if undefinedId newEnv [] (Args []) expId
    then Left $ expId ++ ": undefined symbol"
    else case checkIdEnv newEnv expId of
           Right () -> evalId newEnv expId
           Left _   -> Left $ expId ++ ": invalid arguments"
tryEvalExpression e (ExpList [exp]) = tryEvalExpression e exp
tryEvalExpression e exp@(ExpList ((ExpId expId):expTail)) = do
  newEnv <- checkInferredTypes e [exp]
  newEnv <- return (e `Map.union` newEnv)
  if expId `elem` keywords
  then tryEvalKeyword newEnv exp
  else case forM expTail $ tryEvalExpression newEnv of
         Right expArgs -> case checkCallEnv newEnv $ (ExpId expId):expArgs of
                            Right () -> evalList newEnv $ (ExpId expId):expArgs
                            Left err -> Left $ err ++ ": invalid arguments"
         Left  name    -> Left name
```

Por fim, a função `checkInferredTypes` infere o tipo das expressões sendo executadas e verifica se são válidos no contexto em que se encontram. Por exemplo, considerando que

exista uma função `add`, que some dois números naturais, e uma função `not`, que retorne um valor do tipo `Bool`, ao se tentar executar `add (not true) zero`, a inferência de tipos detectará que se está tentando somar um valor booleano e retornará um erro. Essa verificação é delegada principalmente para a função `bindTypeVars`, mostrada no Quadro 34.

Essa função mantém uma tabela paralela à tabela de variáveis principal do interpretador, em que são mantidos os tipos de valores já inferidos durante a execução de uma expressão. Caso se chegue a tipos incompatíveis, é retornada uma mensagem de erro com o nome da expressão em que se encontrou o problema.

Quadro 34 - Função `bindTypeVars`

```
bindTypeVars binds (TypeId typeId, ExpId expId) =
  case typeId `Map.lookup` binds of
    Just (Var (ExpId expBind)) -> if (expId `Map.lookup` env) == (expBind
`Map.lookup` env)
                                  then return binds
                                  else Left expId
    _                            -> if isAsciiLower $ head typeId
                                  then return (typeId `Map.insert` (Var (ExpId
expId)) $ binds)
                                  else return binds
bindTypeVars binds (DepType name typeExp, ExpList exp@((ExpId expId):expTail)) =
  case expressionSignature expId of
    Just (TypeId typeId)              -> return binds
    Just (DepType expName expTypeExp) -> compareTypeExp binds typeExp expTypeExp
    _                                 -> return binds
bindTypeVars binds (DepType name typeExp, ExpId expId) =
  case expressionSignature expId of
    Just (TypeId typeId)              -> return binds
    Just (DepType expName expTypeExp) -> compareTypeExp binds typeExp expTypeExp
    _                                 -> return binds
bindTypeVars binds (t, e) = return binds
```

3.3.3 Operacionalidade da implementação

Conforme apresentado na definição do caso de uso, existem duas formas pelas quais o usuário pode utilizar o protótipo desenvolvido neste trabalho. Inicialmente, o usuário pode executar o programa, sem nenhum parâmetro adicional, e assim iniciar uma sessão interativa conforme o caso de uso descrito no Quadro 23. Fazendo isso, o usuário se deparará com uma curta mensagem contendo a versão do programa e um *prompt* aguardando sua entrada, como mostra o Quadro 35.

Quadro 35 - Início de uma sessão interativa

```
$ dt-cli
dependent-types v0.1.0.0

>>>
```

O usuário pode, então, começar a digitar comandos na linguagem descrita neste trabalho. Caso o comando atual seja muito grande, o usuário pode continuar a escrevê-lo na próxima linha pressionando a tecla *enter*. O *prompt* mudará de texto, para indicar que essa nova linha é uma continuação do comando anterior, e não um novo comando. O Quadro 36

ilustra essa possibilidade mostrando a definição de um tipo `Bool` em três linhas. Quando se chega ao ponto-final, o interpretador executa o comando que foi escrito e o *prompt* volta a seu formato inicial.

Quadro 36 - Entrada de um comando longo

```
$ dt-cli
dependent-types v0.1.0.0

>>> type Bool : Type where
...   true  : Bool;
...   false : Bool.
>>>
```

A não ser que haja algum erro, comandos para a definição de funções e tipos não imprimem nada na tela senão uma nova linha do *prompt*. Caso o usuário queira executar uma expressão e ver seu resultado, ele deve utilizar um comando de impressão. O Quadro 37 continua a sessão anterior, definindo uma função e mostrando o resultado de um comando `print` que a utiliza.

Quadro 37 - Uso do comando de impressão

```
$ dt-cli
dependent-types v0.1.0.0

>>> type Bool : Type where
...   true  : Bool;
...   false : Bool.
>>> func not : Bool -> Bool where
...   not true  = false;
...   not false = true.
>>> print (not false).
true.
>>>
```

Em casos de erros, independentemente do tipo de comando utilizado, o interpretador mostra uma mensagem contendo o símbolo em que foi encontrado um erro e uma mensagem indicando qual foi o problema. Por exemplo, o Quadro 38 mostra um erro ao se tentar imprimir uma expressão sintaticamente incorreta e outro erro ao se tentar definir uma função para um tipo que ainda não foi definido. Finalmente, o usuário pode digitar `exit.` para encerrar uma sessão interativa e voltar ao *prompt* de seu sistema operacional.

Uma forma alternativa de se utilizar o interpretador desenvolvido é através de arquivos, conforme a extensão do caso de uso descrito no Quadro 23. Para tanto, o usuário escreve os comandos que quer executar em um arquivo de texto e, ao executar o interpretador, passa o nome desse arquivo como parâmetro. Ao fazer isso, o interpretador executará os comandos de forma similar a como os executa em uma sessão interativa: comandos para a definição de tipos e funções não imprimirão nada na tela, somente comandos de impressão. Como exemplo, o conteúdo do Quadro 39 foi gravado em um arquivo chamado `exemplo.dt` e sua execução é demonstrada no Quadro 40.

Quadro 38 - Erros de execução

```
$ dt-cli
dependent-types v0.1.0.0

>>> type Bool : Type where
...   true  : Bool;
...   false : Bool.
>>> func not : Bool -> Bool where
...   not true  = false;
...   not false = true.
>>> print (not false).
true.
>>> print (not (true).
(line 1, column 18):
unexpected "."
expecting space or ")"
>>> func addOne : Nat -> Nat where
...   addOne x = suc x.
Nat: undefined type.
>>> exit.
$
```

Quadro 39 - Arquivo `exemplo.dt`

```
type Nat : Type where
  zero : Nat;
  suc  : Nat -> Nat.

func add : Nat -> Nat -> Nat where
  add zero    y    = y;
  add x       zero = x;
  add (suc x) y    = suc (add x y).

print (add (suc (suc zero)) (suc (suc (suc zero)))).
```

Quadro 40 - Execução do arquivo `exemplo.dt`

```
$ dt-cli exemplo.dt
(suc (suc (suc (suc (suc zero))))).
$
```

Um ponto em que as duas formas de execução diferem é em relação ao tratamento de erros. Em ambos os casos, uma mensagem de erro será impressa. Porém, em uma sessão interativa, o usuário pode digitar novos comandos, enquanto na execução de um arquivo, o interpretador encerra sua execução ao encontrar o primeiro erro.

Outros exemplos de comandos e a descrição da linguagem desenvolvida para este protótipo podem ser encontrados na seção 3.2.2 e no Apêndice A.

3.4 RESULTADOS E DISCUSSÕES

Os resultados obtidos atingiram os objetivos propostos, pois por meio do interpretador desenvolvido podem-se criar programas na linguagem especificada e tê-los executados e validados pela teoria dos tipos dependentes. Desta forma, um programador pode escrever uma função como `pairAdd`, apresentada no Quadro 41 e melhor detalhada no Apêndice A, e o sistema de tipos validará que ela está sempre sendo chamada corretamente, com duas listas de mesmo tamanho. Isso permite que o programador tenha algumas de suas suposições sobre o

programa validadas pelo interpretador, reduzindo o número de verificações que precisa escrever manualmente.

Quadro 41 – Função `pairAdd`

```
func pairAdd : (List n Nat) -> (List n Nat) -> (List n Nat) where
  pairAdd nil         nil         = nil;
  pairAdd (cons x xs) (cons y ys) = cons (add x y) (pairAdd xs ys).
```

Para matemáticos, a utilização da teoria de tipos dependentes permite a validação de teoremas de lógica de predicados construtiva, conforme a equivalência de Howard-Curry (SØRENSEN; URZYCZYN, 2006). Dessa forma, um teorema pode ser codificado através da definição de um tipo. Caso seja possível construir um objeto desse tipo, o teorema é válido, e essa construção serve como uma prova. Por exemplo, o Quadro 42 define dois tipos, `Even` e `Odd`, e prova que a soma de dois números ímpares é par por meio da função `sumOfOdd`.

Quadro 42 – Prova que a soma de dois números ímpares é par

```
type Nat : Type where
  zero : Nat;
  suc  : Nat -> Nat.

func add : Nat -> Nat -> Nat where
  add x    zero = x;
  add zero y    = y;
  add (suc x) y = suc (add x y).

type Even : Nat -> Type where
  evenZero : (Even zero);
  evenSuc  : (Even n) -> (Even (suc (suc n))).

type Odd : Nat -> Type where
  oddOne : (Odd (suc zero));
  oddSuc : (Odd n) -> (Odd (suc (suc n))).

func sumOfOdd : (Odd n) -> (Odd m) -> (Even (add n m)) where
  sumOfOdd oddOne oddOne     = evenSuc evenZero;
  sumOfOdd oddOne (oddSuc y) = evenSuc (sumOfOdd oddOne y);
  sumOfOdd (oddSuc x) oddOne = evenSuc (sumOfOdd x oddOne).
```

Uma limitação do uso da lógica de predicados construtiva é a necessidade da construção de um objeto matemático para provar sua existência. Sendo assim, não é possível desenvolver uma prova por contradição, por exemplo, pois ela não culmina na construção de um desses objetos.

Em comparação aos trabalhos correlatos, o protótipo desenvolvido difere significativamente de todos eles. Com relação a Agda, a linguagem deste trabalho é mais simples, não possuindo meios de se interagir com outras linguagens, uma interface interativa e um sistema de módulos. Comparando-se a Coq, os mesmos pontos podem ser levantados, além de Coq utilizar o cálculo de construções e não só a teoria de tipos dependentes. Por fim, diferentemente do trabalho de Löh, McBride e Swierstra (2010), é possível definir novos tipos sem que seja necessário se basear em um tipo já existente. Além disso, o interpretador desenvolvido é o único que se foca principalmente em programadores com pouco contato com

tipos dependentes. O Quadro 43 mostra essas diferenças em um formato de mais fácil visualização.

Quadro 43 – Comparativo entre os trabalhos correlatos e o protótipo desenvolvido

	Agda	**Coq**	**Löh, McBride, Swierstra (2010)**	**protótipo**
permite a definição de novos tipos	sim	sim	não	sim
permite a interação com outras linguagens	sim	sim	não	não
possui foco em programadores com pouco contato com tipos dependentes	não	não	não	sim
possui interface interativa	sim	sim	não	não
possui sistema de módulos	sim	sim	não	não
utiliza a teoria de tipos dependentes	sim	não	sim	sim

Por último, pode-se comparar a implementação de um algoritmo de ordenação nas linguagens Agda, Coq e a linguagem desenvolvida para este trabalho. Os códigos encontram-se, respectivamente, nos Quadro 15, Quadro 17 e Quadro 51. Para facilitar a comparação, esses quadros são repetidos como Quadro 44, Quadro 45 e Quadro 46. Como justificado na seção 2.5.3, optou-se por não implementar um algoritmo de ordenação na linguagem do trabalho de Löh, McBride e Swierstra (2010) devido às dificuldades de se utilizá-la.

Entre eles, o que se mostrou mais legível foi o desenvolvido em Coq, dada a possibilidade de se escreverem as provas separadas da função em si. Tanto em Agda como na linguagem deste trabalho, há uma complexidade maior, pois a prova deve ser codificada pelos próprios tipos e funções, fazendo com que haja a necessidade de alguns argumentos adicionais. Agda reduz esse problema permitindo a definição de argumentos implícitos, característica não presente na linguagem deste trabalho. Isso elimina a necessidade de usar alguns argumentos quando a função é chamada, desde que eles possam ser inferidos pelo contexto. Porém, ainda que isso simplifique a chamada da função, a definição da função continua necessitando de mais argumentos de que seria necessário em outras linguagens.

Quadro 44 – Trechos do módulo `Sort` em Agda

```
data ⊥XT : Set where
  T ⊥ : ⊥XT
  ⟦_⟧ : X → ⊥XT

data _≤^_ : Rel ⊥XT where
  ⊥≤^     : ∀ {x} → ⊥ ≤^ x
  ≤^T     : ∀ {x} → x ≤^ T
  ≤-lift  : ∀ {x y} → x ≤ y → ⟦ x ⟧ ≤^ ⟦ y ⟧

data OList (l u : ⊥XT) : Set where
  nil  : l ≤^ u → OList l u
  cons : ∀ x (xs : OList ⟦ x ⟧ u) → l ≤^ ⟦ x ⟧ → OList l u

insert : ∀ {l u} x → OList l u → l ≤^ ⟦ x ⟧ → ⟦ x ⟧ ≤^ u → OList l u
  insert y (nil _)         l≤y y≤u = cons y (nil y≤u) l≤y
  insert y (cons x xs l≤x) l≤y y≤u with y ≤? x
  insert y (cons x xs l≤x) l≤y y≤u | left  y≤x = cons y
    (cons x xs (≤-lift y≤x)) l≤y
  insert y (cons x xs l≤x) l≤y y≤u | right y>x =
    cons x (insert y xs ([ ≤-lift , (λ y≤x → absurd (y>x y≤x)) ]
    (total x y)) y≤u) l≤x

isort : List X → OList ⊥ T
  isort = foldr (λ x xs → insert x xs ⊥≤^ ≤^T) (nil ⊥≤^)
```

Fonte: adaptado de Mazzoli (2013).

Quadro 45 – Trechos do módulo `Coq.Sorting.Mergesort` em Coq

```
Fixpoint merge_list_to_stack stack l :=
  match stack with
  | [] => [Some l]
  | None :: stack´ => Some l :: stack´
  | Some l´ :: stack´ => None :: merge_list_to_stack stack´ (merge l´ l)
  end.

Fixpoint merge_stack stack :=
  match stack with
  | [] => []
  | None :: stack´ => merge_stack stack´
  | Some l :: stack´ => merge l (merge_stack stack´)
  end.

Fixpoint iter_merge stack l :=
  match l with
  | [] => merge_stack stack
  | a::l´ => iter_merge (merge_list_to_stack stack [a]) l´
  end.

Definition sort := iter_merge [].

Fixpoint SortedStack stack :=
  match stack with
  | [] => True
  | None :: stack´ => SortedStack stack´
  | Some l :: stack´ => Sorted l /\ SortedStack stack´
  end.

Theorem Sorted_merge_list_to_stack : forall stack l,
  SortedStack stack -> Sorted l -> SortedStack (merge_list_to_stack stack l).
```

Fonte: adaptado de The Coq Development Team (2015b).

Quadro 46 – Função de ordenação com tipos dependentes

```
type LessOrEqual : Nat -> Nat -> Type where
  lessZero zero    y       : Nat -> Nat -> (LessOrEqual zero y);
  lessSuc  (suc x) (suc y) : Nat -> Nat -> (LessOrEqual (suc x) (suc y)) |
LessOrEqual x y.

type Bounded : Type where
  lowerBound : Bounded;
  upperBound : Bounded;
  bounded    : Nat -> Bounded.

{- A lifted version of LessOrEqual with total ordering. -}
type BLessOrEqual : Bounded -> Bounded -> Type where
  bLessLower lowerBound  y           : Bounded -> Bounded ->
                                       (BLessOrEqual lowerBound y);
  bLessUpper x           upperBound  : Bounded -> Bounded ->
                                       (BLessOrEqual x upperBound);
  bLessLift  (bounded x) (bounded y) : Bounded -> Bounded ->
                                       (BLessOrEqual (bounded x) (bounded y)) |
                                        LessOrEqual x y.

type OList : Bounded -> Bounded -> Type where
  onil       : (BLessOrEqual l u) -> (OList l u);
  ocons x xs : Nat -> (OList (bounded x) u) -> (BLessOrEqual l (bounded x)) ->
               (OList l u).

func toList : (OList l u) -> (List n Nat) where
  toList (onil z)       = nil;
  toList (ocons x xs z) = cons x (toList xs).

func insert : Nat ->
              (OList l u) ->
              (BLessOrEqual l (bounded x)) ->
              (BLessOrEqual (bounded x) u) ->
              (OList l u) where

  insert x
         (onil z)
         (BLessOrEqual l (bounded x))
         (BLessOrEqual (bounded x) u) = ocons x
                                            (onil (BLessOrEqual (bounded x) u))
                                            (BLessOrEqual l (bounded x));

  insert x
        (ocons y ys (BLessOrEqual l y))
        (BLessOrEqual l (bounded x))
        (BLessOrEqual (bounded x) u) =

    match (LessOrEqual (suc x) (suc y))
      (ocons x
            (ocons y ys (BLessOrEqual (bounded x) (bounded y)))
            (BLessOrEqual l (bounded x)))
      (ocons y
             (insert x ys (BLessOrEqual (bounded y) (bounded x))
                          (BLessOrEqual (bounded x) u))
             (BLessOrEqual l (bounded y))).
```

4 CONCLUSÕES

No início do trabalho, elencou-se como objetivo criar um provador interativo de teoremas. De forma mais detalhada, objetivava-se especificar uma linguagem, seu sistema de tipos e implementar um interpretador capaz de validar e executar os programas escritos nessa linguagem. Ao término, pode-se verificar que todos os objetivos propostos foram alcançados.

Considerando como público-alvo os programadores, a possibilidade de escrever seus programas nessa linguagem e tê-los verificados por regras de validação mais restritas foi alcançada. No entanto, o uso da teoria de tipos também mostrou alguns inconvenientes. Em particular, devido à necessidade de codificar as provas nos próprios tipos e funções, muitas vezes eles se tornam mais complexos do que seriam em uma linguagem sem tipos dependentes. Um sistema de provas em que elas possam ser escritas à parte da função em si, como no caso de Coq, provavelmente seria mais vantajoso.

Para matemáticos, permitiu-se a escrita de teoremas de lógica de predicados construtiva. Para isso, o teorema que se deseja provar deve ser codificado como um tipo e a construção de um valor desse tipo cumpre o papel de uma prova. Porém, a lógica construtiva também possui suas limitações. Devido à necessidade de se construir um objeto lógico para que se considere uma prova como válida, outros tipos de prova, tal como a prova por contradição, não são considerados válidos dentro desse contexto.

A escolha de ferramentas para o desenvolvimento do trabalho foi bem acertada. A linguagem Haskell mostrou-se bastante expressiva, permitindo a escrita do interpretador em poucas linhas de código, e não houve problemas com a implementação escolhida (o compilador GHC). A escolha de linguagem também possibilitou a utilização da biblioteca Parsec, que permitiu a escrita do analisador sintático de maneira sucinta e muito próxima à notação utilizada na especificação da gramática. A ferramenta Cabal foi útil no gerenciamento do projeto, instalando as dependências necessárias e facilitando o processo de compilar, testar e executar o interpretador. A utilização do HUnit também foi um facilitador, automatizando o processo de testes, fazendo com que não fosse necessário executar diversos casos de testes para cada alteração feita no protótipo.

A única ferramenta que não teve grande impacto durante a elaboração do trabalho foi o Haddock, utilizada para a documentação. Embora os módulos e as funções tenham sido documentados segundo sua notação sem problemas, não houve utilidade prática para a documentação gerada a partir delas. No entanto, espera-se que seja útil no desenvolvimento de extensões para este trabalho.

4.1 EXTENSÕES

Embora o trabalho tenha atingido seu objetivo, existem possibilidades de aprimorá-lo e incrementá-lo, sendo elas:

a) desenvolver uma interface gráfica ou um modo interativo, como Agda disponibiliza para Emacs;
b) acrescentar um sistema de módulos para a linguagem desenvolvida;
c) extrair o código fonte para outras linguagens já existentes, como Coq consegue fazer para OCaml, por exemplo;
d) utilizar a linguagem deste trabalho para desenvolver programas certificados, conforme descreve Chlipala (2013);
e) possibilitar a geração de executáveis através de um compilador em vez de utilizar sempre o interpretador.

REFERÊNCIAS

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

ASPERTI, Andrea. **A survey on interactive theorem proving**. [S.l.], 2009. Disponível em: <http://www.cs.unibo.it/~asperti/SLIDES/itp.pdf>. Acesso em: 24 maio 2015.

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.

BARENDREGT, Henk; BARENDSEN, Erik. **Introduction to Lambda Calculus**. [Nijmegen], 2000. Disponível em: <http://ftp.cs.ru.nl/CompMath.Found/lambda.pdf>. Acesso em: 15 fev. 2015.

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.

COOK, William R.; HILL, Walter; CANNING, Peter S. Inheritance is not subtyping. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 17, 1989, San Francisco. **Proceedings...**. New York: ACM, 1990. p. 125 - 135.

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.

DEVRIESE, Dominique. **The Agda wiki**: foreign function interface. [S.l.], 2014. Disponível em: <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.ForeignFunctionInterface>. Acesso em: 10 maio 2015.

FERNANDES, Fabiana L. **O isomorfismo de Curry-Howard via teoria das categorias**. 2009. 79 f. Dissertação (Mestrado em Matemática), Universidade Federal de Minas Gerais, Belo Horizonte. Disponível em: <http://www.mat.ufmg.br/intranet-atual/pgmat/TesesDissertacoes/uploaded/Diss166.pdf>. Acesso em: 24 maio 2015.

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.

HERINGTON, Dean. **HUnit**: a unit testing framework for Haskell. [S.l.], 2013. Disponível em: <https://hackage.haskell.org/package/HUnit>. Acesso em: 23 maio 2015.

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. **A tutorial implementation of a dependently typed Lambda calculus**: Haskell source code. [S.l.], 2009a. Disponível em: <http://www.andres-loeh.de/LambdaPi/LambdaPi.hs>. Acesso em: 10 maio 2015.

_____. **A tutorial implementation of a dependently typed Lambda calculus**: prelude. [S.l.], 2009b. Disponível em: <http://www.andres-loeh.de/LambdaPi/prelude.lp>. Acesso em: 10 maio 2015.

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.

MARLOW, Simon (Org.). **Haddock**. [S.l.], 2014. Disponível em: <https://www.haskell.org/haddock/>. Acesso em: 23 maio 2015.

_____. **The Glasgow Haskell Compiler**. [S.l.], 2015. Disponível em: <https://www.haskell.org/ghc/>. Acesso em: 23 maio 2015.

MARTIN-LÖF, Per. Constructive mathematics and computer programming. In: DISCUSSION MEETING OF THE ROYAL SOCIETY OF LONDON ON MATHEMATICAL LOGIC AND PROGRAMMING LANGUAGES, 1., 1982, London. **Proceedings...** . Upper Saddle River: Prentice-hall, 1985. p. 167-184. Disponível em: <http://www.cs.tufts.edu/~nr/cs257/archive/per-martin-lof/constructive-math.pdf>. Acesso em: 09 maio 2015.

MAYDWELL, Lyndon. **A simple introduction to Agda**. [S.l.], 2012. Disponível em: <https://www.youtube.com/watch?v=pP7ynVdVY9A>. Acesso em: 10 maio 2015.

MAZZOLI, Francesco. **Agda by example**: sorting. [S.l.], 2013. Disponível em: <http://mazzo.li/posts/AgdaSort.html>. Acesso em: 09 maio 2015.

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.

_____. 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.

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.

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.

O'SULLIVAN, Bryan; STEWART, Don; GOERZEN, John. **Real world Haskell**: code you can believe in. [S.l.]: O'Reilly Media, 2008.

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, 1st, 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 CABAL DEVELOPMENT TEAM. **The Haskell Cabal**. [S.l.], 2015. Disponível em: <https://www.haskell.org/cabal/>. Acesso em: 23 maio 2015.

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

_____. **The Coq proof assistant**: Coq integrated development environment. [S.l.], 2015a. Disponível em: <https://coq.inria.fr/distrib/current/refman/Reference-Manual018.html>. Acesso em: 10 maio 2015.

_____. **The Coq proof assistant**: standard library. [S.l.], 2015b. Disponível em: <https://coq.inria.fr/library/Coq.Sorting.Mergesort.html>. Acesso em: 10 maio 2015.

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.

APÊNDICE A – Exemplo de código escrito na linguagem do protótipo

A seguir, apresentam-se exemplos de código escritos na linguagem desenvolvida para o protótipo. O Quadro 47 mostra algumas operações simples, como a definição de tipos e constantes. Começa-se com um comentário, um texto escrito entre os caracteres `{-` e `-}`. Em seguida, define-se um tipo simples chamado `Bool`, cuja assinatura é `Type` e que possui dois construtores, `true` e `false`. Por sua vez, esses construtores possuem assinatura `Bool`.

Depois, define-se outro tipo, `Nat`, também com assinatura `Type` e com construtores `zero` e `suc`. Diferentemente do tipo `Bool`, que pode ser construído com qualquer uma das duas constantes `true` ou `false`, `Nat` só pode ser construído com base na constante `zero`. O construtor `suc`, cuja assinatura é `Nat -> Nat`, é utilizado para construir o sucessor de outro valor do tipo `Nat`. Assim, os números naturais podem ser representados indutivamente através de `zero`, `suc zero`, `suc (suc zero)` e assim por diante.

Então, é criada uma função chamada `six`, cuja assinatura é `Nat`. Como essa função não aceita argumentos e retorna sempre o mesmo valor, ela pode ser compreendida como uma constante. Da mesma forma, a função definida posteriormente, `seven`, também pode ser compreendida como constante.

Por fim, utilizam-se dois comandos de impressão. O primeiro imprime as constantes definidas para o tipo `Bool` e o segundo imprime o valor das constantes `six` e `seven`, definidas com base no tipo `Nat`.

Quadro 47 – Definições de tipos e constantes

```
{- Uma forma como booleanos podem ser descritos. -}
type Bool : Type where
  true  : Bool;
  false : Bool.

{- Uma forma como números naturais podem ser descritos. -}
type Nat : Type where
  zero : Nat;
  suc  : Nat -> Nat.

{- Definição da função six, que é uma constante. -}
func six : Nat where
  six = suc (suc (suc (suc (suc (suc zero))))).

{- Definição da função seven, que também é uma constante. -}
func seven : Nat where
  seven = suc six.

print true; false.
print six; seven.
```

Conforme exibe o Quadro 48, funções com argumentos são definidas de forma similar às constantes do quadro anterior. Quando se define uma função que aceita argumentos, sua assinatura passa a ser composta, como é o caso da função `not`, cuja assinatura é `Bool ->`

`Bool`. Isso significa que esta função aceita um argumento do tipo `Bool` e retorna um valor também do tipo `Bool`.

Outra característica da linguagem é o uso de casamento de padrões. Ao se definir uma função, podem-se estipular diferentes padrões que funcionam de forma similar a condicionais. Assim, quando a função `not` for chamada com `true` como parâmetro, o resultado será `false`. De forma similar, quando `not` for chamada com `false`, o resultado será `true`.

Processo similar é realizado quando a função `add` é chamada. Quando o segundo argumento é `zero`, é chamada a primeira implementação. Da mesma forma, quando o primeiro argumento é `zero`, é chamada a segunda implementação. Por fim, se nenhuma das possibilidades anteriores for verdade, aplica-se o terceiro caso.

No Quadro 48 também é apresentado um exemplo de funções mutuamente recursivas, `isOdd` e `isEven`. A definição desse tipo de função não é muito diferente das demais funções, só é necessário que o nome e a assinatura dessas funções sejam especificados em conjunto, separados pelo símbolo ponto-e-vírgula.

Quadro 48 – Definições de funções com argumentos

```
{- Definição da função negação. -}
func not : Bool -> Bool where
  not true  = false;
  not false = true.

{- Definição da função soma. -}
func add : Nat -> Nat -> Nat where
  add x       zero = x;
  add zero    y    = y;
  add (suc x) y    = suc (add x y).

{- Definição de funções mutuamente recursivas. -}
func isOdd : Nat -> Bool; isEven : Nat -> Bool where
  isOdd zero     = false;
  isEven zero    = true;
  isOdd (suc x)  = isEven x;
  isEven (suc x) = isOdd x.
```

Tanto na terceira implementação da função `add` como nas últimas implementações das funções `isOdd` e `isEven` é utilizada mais uma característica da linguagem, a desconstrução de argumentos. Quando um argumento é especificado com um construtor que, por sua vez, também recebe argumentos, a associação de variáveis desconsidera a parte comum à especificação do argumento e ao valor passado como argumento. Por exemplo, caso se chame `isOdd` com o valor `suc (suc (suc zero))`, detecta-se que o argumento `suc x` também começa com `suc`, elimina-se esse primeiro `suc` do valor, e atribui-se apenas `suc (suc zero)` para o valor de `x`. O Quadro 49 ilustra o processo com alguns valores.

Quadro 49 – Desconstrução de argumentos

definição do argumento	valor do argumento	valor de x
`suc x`	`suc zero`	`zero`
`suc x`	`suc (suc zero)`	`suc zero`
`suc x`	`suc (suc (suc zero))`	`suc (suc zero)`

Posteriormente, no Quadro 50, define-se o tipo `List`. Diferentemente dos exemplos anteriores, sua assinatura é composta. Assim, para se especificar um valor do tipo `List`, é necessário também especificar um número natural (seu tamanho) e o tipo dos elementos que compõem essa lista. Por exemplo, `List zero Bool` representa uma lista de valores do tipo `Bool` vazia, enquanto `List (suc zero) Nat` representa uma lista com apenas um elemento do tipo `Nat`.

Esse tipo pode ser construído com base na lista vazia, representada por `nil` e cujo tamanho indicado na assinatura é `zero`. Para inserir elementos em uma lista, utiliza-se o construtor `cons`, que recebe um elemento e uma lista já existente como argumentos e retorna uma lista com um elemento a mais. Isso é representado na assinatura do construtor através da variável `n`. A lista utilizada como argumento possui um tamanho qualquer, `n`, e a lista retornada possui tamanho `suc n`.

A função `pairAdd` demonstra uma das vantagens de se utilizar tipos dependentes. Como o tamanho de uma lista é armazenado como parte de seu tipo, o próprio sistema de tipos é capaz de validar se a função está sendo chamada com os argumentos corretos, isso é, com duas listas de mesmo tamanho. Em linguagens sem tipos dependentes, essa verificação teria de ser feita manualmente pelo programador. Além disso, o sistema de tipos também verifica se as listas passadas como argumentos contêm elementos do tipo `Nat`, embora essa última validação também seja realizada em sistemas de tipos sem tipos dependentes, mas com polimorfismo.

Quadro 50 – Utilização de tipos dependentes

```
{- A forma como listas podem ser descritas, utilizando o tamanho dela
 - como parte do tipo. -}
type List : Nat -> Type -> Type where
  nil  : (List zero a);
  cons : a -> (List n a) -> (List (suc n) a).

{- Definição da função tamanho. -}
func length : (List n a) -> Nat where
  length list = n.

{- Função soma par a par.  Tentar chamá-la com listas de tamanhos
 - diferentes é um erro. -}
func pairAdd : (List n Nat) -> (List n Nat) -> (List n Nat) where
  pairAdd nil         nil         = nil;
  pairAdd (cons x xs) (cons y ys) = cons (add x y) (pairAdd xs ys).
```

Finalmente, o Quadro 51 ilustra um caso mais complexo, uma lista obrigatoriamente ordenada, adaptada da implementação de Mazzoli (2013) para Agda. Inicia-se a definição

com um tipo capaz de representar a ordenação dos números naturais, `LessOrEqual`. Seus dois construtores são `lessZero`, representando que `zero` é menor ou igual a todos os outros números naturais, e `lessSuc`, representando os outros casos. Para esse segundo construtor, é utilizada uma restrição, que trabalha recursivamente até chegar a um caso que possa ser construído por `lessZero`. Por exemplo, `lessSuc (suc (suc zero)) (suc (suc (suc zero)))` implica que `lessSuc (suc zero) (suc (suc zero))` seja verdade. Por sua vez, isso implica que `lessZero zero (suc zero)` seja uma construção válida, o que é correto. Porém, caso se tente mostrar a validade de `lessSuc (suc (suc zero)) (suc zero)`, chega-se a `lessSuc (suc zero) zero`, que é uma construção inválida.

Em seguida, define-se o tipo `Bounded`, que pode ser compreendido como uma extensão aos números naturais, acrescentando um limite mínimo (`lowerBound`) e um limite máximo (`upperBound`). Para trazer um valor do tipo `Nat` para o tipo `Bounded`, é utilizado o construtor `bounded`.

Então, cria-se uma versão de `LessOrEqual` aplicável a valores do tipo `Bounded`, chamada de `BLessOrEqual`. O valor `lowerBound` é menor que todos os outros valores do tipo `Bounded`, como representado pelo construtor `bLessLower`, enquanto que `upperBound` é maior que todos, representado por `bLessUpper`. Por fim, a comparação entre números naturais que foram trazidos para o tipo `Bounded` pode ser feita com o construtor `bLessLift`.

Finalmente, especifica-se um tipo `OList`, que representa uma lista cujos elementos estão sempre ordenados, e uma função `insert`, responsável por inserir os valores na lista de forma ordenada. Embora aqui se demonstre outra vantagem do uso de tipos dependentes, a possibilidade de verificar se um valor é menor que outro ou não através de seu tipo, também fica claro que em algumas situações isso aumenta a complexidade do código sendo escrito. Por exemplo, a função `insert`, além de receber o elemento a ser inserido e a lista em que ele será inserido, necessita de alguns argumentos adicionais do que se esperaria numa linguagem sem tipos dependentes.

Quadro 51 – Função de ordenação com tipos dependentes

```
type LessOrEqual : Nat -> Nat -> Type where
  lessZero zero    y       : Nat -> Nat -> (LessOrEqual zero y);
  lessSuc  (suc x) (suc y) : Nat -> Nat -> (LessOrEqual (suc x) (suc y)) |
LessOrEqual x y.

type Bounded : Type where
  lowerBound : Bounded;
  upperBound : Bounded;
  bounded    : Nat -> Bounded.

{- A lifted version of LessOrEqual with total ordering. -}
type BLessOrEqual : Bounded -> Bounded -> Type where
  bLessLower lowerBound  y           : Bounded -> Bounded ->
                                       (BLessOrEqual lowerBound y);
  bLessUpper x           upperBound  : Bounded -> Bounded ->
                                       (BLessOrEqual x upperBound);
  bLessLift  (bounded x) (bounded y) : Bounded -> Bounded ->
                                       (BLessOrEqual (bounded x) (bounded y)) |
                                        LessOrEqual x y.

type OList : Bounded -> Bounded -> Type where
  onil       : (BLessOrEqual l u) -> (OList l u);
  ocons x xs : Nat -> (OList (bounded x) u) -> (BLessOrEqual l (bounded x)) ->
               (OList l u).

func toList : (OList l u) -> (List n Nat) where
  toList (onil z)       = nil;
  toList (ocons x xs z) = cons x (toList xs).

func insert : Nat ->
              (OList l u) ->
              (BLessOrEqual l (bounded x)) ->
              (BLessOrEqual (bounded x) u) ->
              (OList l u) where

  insert x
         (onil z)
         (BLessOrEqual l (bounded x))
         (BLessOrEqual (bounded x) u) = ocons x
                                            (onil (BLessOrEqual (bounded x) u))
                                            (BLessOrEqual l (bounded x));

  insert x
        (ocons y ys (BLessOrEqual l y))
        (BLessOrEqual l (bounded x))
        (BLessOrEqual (bounded x) u) =

    match (LessOrEqual (suc x) (suc y))
      (ocons x
            (ocons y ys (BLessOrEqual (bounded x) (bounded y)))
            (BLessOrEqual l (bounded x)))
      (ocons y
             (insert x ys (BLessOrEqual (bounded y) (bounded x))
                          (BLessOrEqual (bounded x) u))
             (BLessOrEqual l (bounded y))).


print (toList (insert (suc zero)
                      (insert zero
                              (onil (bLessLower lowerBound upperBound))
                              (bLessLower lowerBound (bounded zero))
                              (bLessUpper (bounded zero) upperBound)))
                      (bLessLower lowerBound (bounded (suc zero)))
                      (bLessUpper (bounded (suc zero)) upperBound))).
```