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

LEONARDO BRANCHI NASCIMENTO

Programação com tipos dependentes: uma revisão

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

Orientador: Prof. Dr. Álvaro Freitas Moreira

Porto Alegre
2025

“Vibra o Brasil inteiro com o clube do povo do Rio Grande do Sul.”

— NELSON SILVA

AGRADECIMENTOS

Agradeço a Deus, sem o qual nada é possível.

Agradeço à minha família pelo apoio em toda a minha vida.

Agradeço aos meus colegas do curso de Ciência da Computação pelo convívio e companheirismo durante todos esses anos.

Por fim, agradeço ao meu orientador, Álvaro Freitas Moreira, que me aceitou como orientando e me acompanhou durante o desenvolvimento deste trabalho.

RESUMO

Tipos dependentes são tipos cuja definição depende de algum valor, e estão comumente presentes em linguagens de programação com foco em prova de teoremas. O objetivo deste trabalho é a realização de uma revisão sistemática sobre o uso de tipos dependentes, com ênfase em seu uso para programação convencional.

Palavras-chave: Tipos dependentes. Prova de teoremas. Lean. Idris. Agda.

Programming with dependent types: a systematic revision

ABSTRACT

Dependent types are types whose definition depends on a value, and they are commonly present in programming languages focused on theorem proving. The objective of this work is the realization of a systematic revision regarding the usage of dependent types, focusing on its usage for conventional programming.

Keywords: Dependent types. Theorem proving. Lean. Idris. Agda..

LISTA DE ABREVIATURAS E SIGLAS

IPL	Intuitionistic propositional logic (lógica proposicional intuicionista)
STLC	Simply-typed lambda calculus (cálculo lambda simplesmente tipado)

SUMÁRIO

1 INTRODUÇÃO

Sistemas de tipos são uma parte fundamental da programação, auxiliando a evitar erros e aumentando a legibilidade e manutenibilidade do código. De acordo com Pierce (2002), "um sistema de tipos é um método sintático tratável para provar a ausência de certos comportamentos de programa ao classificar frases de acordo com a espécie de valores que computam". Além de auxiliar na prevenção de erros, tipos também são uma poderosa ferramenta de abstração, aumentando a modularidade e facilitando o reuso de código.

Como veremos adiante, há uma equivalência fundamental entre computação e lógica, na qual tipos correspondem a proposições e programas correspondem a provas. Desta maneira, tipos suficientemente expressivos, como tipos dependentes, podem ser usados para provar formalmente que um programa se comporta da maneira esperada.

Em sua versão irrestrita, tipos dependentes eliminam a distinção entre tipos e termos e permitem que expressões arbitrárias sejam avaliadas em tempo de compilação. Isso permite que um tipo expresse propriedades desejáveis de um programa. Nesse contexto, o tipo atua como uma especificação formal do programa, e um programa que seja aprovado pela verificação de tipos é um programa correto em relação à especificação expressa no tipo.

Apesar de poderosos, o uso de tipos dependentes vem com vários desafios. Um sistema de tipos com uma versão totalmente irrestrita de tipos dependentes é necessariamente indecidível, pois é impossível garantir a terminação da verificação de tipos. Além disso, tipos dependentes também possuem uma grande dificuldade de usabilidade por programadores, geralmente tendo uma sintaxe complexa e exigindo um conhecimento adequado de aspectos formais de programação e de lógica para um uso efetivo.

O objetivo central deste trabalho é a realização de um estudo compreensivo sobre o uso prático de tipos dependentes em programação de propósito geral, com o fornecimento de exemplos de uso. Contudo, para a devida contextualização, será feita também uma análise histórica dos tipos dependentes, tanto de sua definição formal quanto de outros usos possíveis.

1.1 Estrutura do trabalho

O Capítulo 2 apresentará uma perspectiva histórica dos tipos dependentes, começando por uma breve definição, abordando a história do isomorfismo de Curry-Howard e sua relação com tipos dependentes, detalhando o uso de tipos dependentes para a prova de teoremas e programação de propósito geral e finalizando com uma análise do impacto dos tipos dependentes na decidibilidade da verificação de tipos.

O Capítulo 3 apresentará uma análise das linguagens de programação que implementam tipos dependentes, começando pelas linguagens mais antigas encontradas. Será também abordado o provador de teoremas Coq, as linguagens Agda, Idris, Lean e F* e finalizando com linguagens que possuem versões restritas ou que permitem simular tipos dependentes.

O Capítulo 4 fará um breve comentário sobre a maneira como a pesquisa para este trabalho foi realizada, passando na sequência para exemplos de programas ou funções que utilizem tipos dependentes.

Por fim, o Capítulo 5 será a conclusão do trabalho, passando por uma análise crítica sobre o uso de tipos dependentes em programação de propósito geral e finalizando com propostas de trabalhos futuros.

2 HISTÓRIA E DEFINIÇÃO DE TIPOS DEPENDENTES

Este capítulo começará com uma breve explicação informal sobre tipos dependentes, passando para uma explicação sobre o isomorfismo de Curry-Howard e sua relação com tipos dependentes. Também será abordada a importância dos tipos dependentes para a prova de teoremas e a possibilidade de seu uso em programação de propósito geral, finalizando com uma análise do impacto dos tipos dependentes na decidibilidade da verificação de tipos.

2.1 Breve definição informal

De maneira informal, é possível definir um tipo dependente como um tipo cuja definição depende de uma expressão da linguagem. A título de exemplo em pseudocódigo, utilizando uma sintaxe próxima a linguagens como C# ou Java, uma possível definição de um tipo para um vetor de tamanho fixo seria `Vector<T: Type, e:int>`, em que `T` é um tipo da linguagem e deve ser o tipo dos elementos do vetor e `e` é uma expressão do tipo `int`, que corresponde ao comprimento do vetor. Alguns tipos concretos possíveis para a definição acima são `Vector<int, 3>` e `Vector<string, 4+2>`.

Em uma linguagem que implemente programação genérica, um tipo como, por exemplo, `List<T:Type>` pode ser compreendido como um tipo que depende de outro tipo, ou seja, construtor de tipos que retorna um tipo concreto para cada tipo `T` da linguagem. De maneira equivalente, é possível compreender `Vector` como um tipo que depende de uma expressão, ou seja, um construtor de tipos que recebe um tipo `T` e um inteiro `e` e constrói um tipo concreto.

2.2 Isomorfismo de Curry-Howard e a descoberta dos tipos dependentes

É possível compreender o isomorfismo de Curry-Howard de maneira intuitiva a partir de duas afirmações: tipos são proposições e programas são provas. A partir dessas duas afirmações, é possível criar equivalências entre os demais elementos da lógica e da computação.

A título de exemplo, é possível afirmar que a regra da eliminação da implicação em IPL (lógica proposicional intuicionista) é equivalente à aplicação lambda em STLC

(cálculo lambda simplesmente tipado):

$$\frac{\Gamma \vdash \phi \implies \psi \quad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

é equivalente a

$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \quad \Gamma \vdash N : \tau_1}{\Gamma \vdash MN : \tau_2}$$

Também é possível afirmar que a regra de introdução da conjunção em IPL

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

é equivalente à regra de tipo de um par ordenado em STLC:

$$\frac{\Gamma \vdash M : \tau_1 \quad \Gamma \vdash N : \tau_2}{\Gamma \vdash \langle M, N \rangle : \tau_1 \times \tau_2}$$

Haskell Curry foi o primeiro a perceber uma aparente equivalência entre construções da lógica intuicionista proposicional e o cálculo combinatório (Curry, 1934), contudo sem avançar nas consequências desta equivalência. William Alvin Howard foi o primeiro a descrever explicitamente a equivalência entre proposições e tipos (Howard, 1980), abrindo caminho para a interpretação moderna do isomorfismo de Curry-Howard.

Per Martin-Löf criou a teoria dos tipos intuicionista e definiu os tipos-Π e tipos-Σ, formalizando o conceito de tipos dependentes, embora não tenha usado essa expressão (Martin-Löf, 1975). Os tipos-Π e tipos-Σ são o equivalente da teoria de tipos a quantificadores universais e existenciais do cálculo de predicados, respectivamente.

Thierry Coquand criou o cálculo de construções, um formalismo extremamente poderoso criado à luz do isomorfismo de Curry-Howard com o objetivo explícito de ser uma "teoria unificadora"da teoria dos tipos e de servir como base para a prova de teoremas e linguagens de programação (Coquand; Huet, 1986).

2.3 Tipos dependentes e prova de teoremas

De acordo com o isomorfismo de Curry-Howard, tipos são proposições e programas são provas. A consequência imediata dessas afirmações é tornar possível a prova de teoremas apenas pela implementação de um programa em uma linguagem de programação adequada, que seja do tipo equivalente à proposição que se deseja provar.

A formulação original do isomorfismo de Curry-Howard é específica a IPL. Contudo, é essencial que um provador de teoremas implemente tipos dependentes, uma vez que permitem que a linguagem associada ao provador de teoremas expresse construções do cálculo de predicados, que é certamente o mais usado em provas tanto na matemática quanto na computação. Sem tipos dependentes, o provador estaria limitado a provar teoremas do cálculo proposicional (ou de outro formalismo menos expressivo e que não possua quantificadores universais e existenciais).

2.4 Tipos dependentes em programação de propósito geral

Tipos dependentes também podem ser usados para programação de propósito geral. Contudo, as implementações mais expressivas de tipos dependentes parecem estar disponíveis somente em linguagens de programação cujo foco é a prova de teoremas (como Idris e Agda) ou em linguagens experimentais (como as extensões à linguagem Haskell fornecidas pelo compilador GHC), que são linguagens com pouca tração fora do contexto de prova de teoremas.

Há várias linguagens mais populares que possuem versões mais restritas de tipos dependentes ou que possibilitam simular tipos dependentes com outros recursos da linguagem. No decorrer do desenvolvimento deste trabalho, foi possível perceber que essas versões restritas de tipos dependentes frequentemente não são descritas na documentação das linguagens, ou são descritas sem mencionar que se tratam de tipos dependentes.

2.5 Tipos dependentes e a decidibilidade da verificação de tipos

Um sistema de tipos para uma linguagem de programação só é viável na prática se o problema de verificação de tipos para ele for decidível, ou seja, se existe algoritmo que termina e responde se o programa é bem-tipado ou não.

Sistemas de tipos que não possuem tipos dependentes são tipicamente decidíveis, sendo possível implementar um algoritmo que faça a verificação de tipos usando apenas informações sintáticas presentes na estrutura sintática dos tipos e do programa, não sendo necessária a avaliação de qualquer expressão.

A implementação de uma versão irrestrita de tipos dependentes faz com que um tipo possa depender de uma expressão arbitrária da linguagem, trazendo toda a comple-

xidade da linguagem para dentro do sistema de tipos. E, de acordo com o problema da parada, se a linguagem é Turing-completa, não é possível construir um algoritmo que determine se uma expressão qualquer da linguagem termina para todas as entradas possíveis. Isso traz a possibilidade de um tipo depender de uma expressão que não termina, fazendo com que o verificador de tipos entre em *loop*.

Para linguagens de programação que implementam uma versão irrestrita de tipos dependentes, a técnica mais comum para contornar esse problema é o uso de estratégias parciais de verificação de tipos, exigindo que o programador forneça uma prova ou anotação para os casos em que o verificador de tipos não foi capaz de garantir que o programa é bem-tipado.

3 LINGUAGENS DE PROGRAMAÇÃO COM TIPOS DEPENDENTES

O objetivo deste capítulo é fornecer uma visão geral e histórica das linguagens que possuem suporte a tipos dependentes, detalhando suas funcionalidades e limitações, o status de desenvolvimento e nível de suporte.

3.1 Primeiras linguagens de programação com tipos dependentes

Se considerarmos versões restritas de tipos dependentes, é possível afirmar que a presença de tipos dependentes em linguagens de programação é tão antiga quanto as primeiras linguagens de programação de alto nível. Fortran e Pascal permitem que o tamanho do array seja especificado na definição da variável e validam a atribuição inicial em tempo de compilação, porém aceitam apenas expressões resolvíveis em tempo de compilação e não utilizam a informação de comprimento para validar acessos posteriores ao array. Também não é possível para o programador definir novos tipos dependentes.

A mais antiga linguagem com uma implementação mais expressiva de tipos dependentes encontrada no desenvolvimento deste trabalho é Automath. Criada na década de 60 por N.G. de Bruijn, teve como objetivo principal a expressão de teoremas matemáticos com o objetivo de automatizar a verificação de provas (Bruijn, 1994). Embora, pelo isomorfismo de Curry-Howard, seja possível utilizar a linguagem para programação, a linguagem foi criada com o foco em construção de provas, não possuindo conveniências que são comuns em linguagens modernas. A linguagem possui uma implementação de tipos dependentes com expressividade intermediária: um tipo pode depender de expressões constantes ou de variáveis definidas anteriormente. Automath teve pouca divulgação à época e, embora tenha servido de inspiração para provadores de teoremas, teve adoção restrita. Atualmente, a linguagem se encontra abandonada.

A mais antiga linguagem de programação que implementa uma versão completa de tipos dependentes encontrada no desenvolvimento deste trabalho é a linguagem utilizada no assistente de provas NuPRL (Constable et al., 1985). Com sua primeira versão lançada em 1984, a linguagem foi projetada para ser usada exclusivamente com um assistente de provas. Embora possua adoção limitada, a linguagem ainda recebe atualizações esporádicas pelo PRL Project da Universidade de Cornell.

3.2 O provador de teoremas Coq

A primeira versão do provador de teoremas Coq foi lançada em 1989, tendo como base o cálculo de construções (The Coq Development Team, 2024). Coq obteve grande adoção na comunidade matemática, sendo usado para a automatização de diversas provas, entre elas uma prova mais acessível para o teorema das quatro cores em 2005 (Gonthier et al., 2008).

Apesar do foco em prova de teoremas, Coq foi usada para a implementação de alguns programas e bibliotecas de programação de propósito geral. Um uso de destaque é o CompCert, um compilador de C – mais especificamente, um subconjunto quase completo de C99 – escrito e verificado formalmente em Coq (Leroy et al., 2016).

Para garantir a decidibilidade da verificação de tipos, a linguagem exige normalização fraca. Para expressões que o verificador de tipos não consegue determinar a terminação, é necessário que o programador forneça prova explícita.

3.3 Demais linguagens com tipos dependentes e programação de propósito geral

Ao longo dos anos, diversas outras linguagens com ênfase em prova de teoremas foram lançadas, porém a maioria se manteve como curiosidade acadêmica e não obteve nível de adoção próximo ao do Coq.

Em 1998, Hongwei Xi e Frank Pfenning propuseram uma extensão à linguagem ML com uma versão restrita de tipos dependentes (Xi; Pfenning, 1998), tendo o objetivo principal de representar o comprimento de arrays no sistema de tipos e evitar a verificação de limites do array em tempo de execução. Uma ideia semelhante foi proposta por Condit et al. (2007) com o sistema de tipos Deputy (Condit et al., 2007), uma extensão para C com o objetivo principal de permitir a verificação de índices de array em tempo de compilação para aplicações de baixo nível.

Agda

Em 1999, foi lançada a primeira versão de Agda, uma linguagem funcional com uma implementação expressiva de tipos dependentes. Embora a linguagem também permita a prova de teoremas e este seja o seu principal uso na comunidade acadêmica, o desenvolvimento da linguagem e a sua documentação dão ênfase na programação de propósito geral. Desta maneira, foi a mais antiga linguagem encontrada no desenvolvimento

deste trabalho que possui suporte oficial a programação de propósito geral e implementa uma versão irrestrita de tipos dependentes (Norell, 2007).

Agda possui ampla biblioteca padrão com diversos tipos comuns implementados usando tipos dependentes. Contudo, a linguagem não parece ter muita tração fora da comunidade matemática, uma vez que não encontramos exemplos de bibliotecas ou sistemas desenvolvidos em Agda.

A linguagem Agda exige normalização forte para todas as expressões usadas no sistema de tipos, garantindo que são terminantes. É possível marcar uma expressão como terminante e forçar seu uso em um tipo dependente, correndo o risco do verificador de tipos entrar em *loop* ou se tornar logicamente inconsistente.

Idris

Idris é uma linguagem funcional que implementa tipos dependentes de maneira quase compeleta. Com a sua primeira versão sendo lançada em 2007, a linguagem dá ênfase para a programação de propósito geral (Brady, 2017). A linguagem é compilada para C ou JavaScript, e há geradores de código alternativos que compilam código em Idris para outras plataformas.

Apesar da ênfase em programação, a linguagem ainda tem um ecossistema pouco desenvolvido, com uma biblioteca padrão relativamente enxuta. Não encontramos exemplos de bibliotecas ou sistemas desenvolvidos em Idris.

A linguagem Idris aceita que apenas expressões comprovadamente terminantes sejam usadas em tipos dependentes. É possível adicionar uma anotação para forçar o uso de expressões potencialmente não-terminantes em tipos dependentes, porém correndo o risco de comprometer a consistência do sistema de tipos ou do verificador de tipos entrar em *loop*.

Lean

Lean é uma linguagem que implementa tipos dependentes com poucas restrições. Teve a sua primeira versão lançada em 2013, com o seu desenvolvimento inicialmente feito a partir da Microsoft Research (Christiansen, David Thrane, 2023). A linguagem foi projetada tanto com prova de teoremas quanto programação em mente, tendo documentação distinta para ambos os casos de uso.

A linguagem está crescendo em popularidade para a prova de teoremas, mas ainda possui pouca adoção na comunidade de desenvolvimento. A versão 4 da linguagem foi

lançada em 2021, consistindo em uma reescrita completa da linguagem, adicionando a possibilidade de o código em Lean ser compilado para C, entre outras melhorias para o uso em programação. Uma característica marcante da linguagem é a multiplicidade de sintaxes, facilitando o aprendizado tanto por programadores quanto por matemáticos.

Lean só permite que expressões comprovadamente terminantes sejam usadas em tipos dependentes, sendo necessário fornecer prova explícita de terminação para demais casos.

F*

F* é uma linguagem multi-paradigma com uma implementação expressiva de tipos dependentes. A linguagem teve sua primeira versão lançada em 2011 e seu desenvolvimento é patrocinado pela Microsoft Research e pelo INRIA (instituto público francês de pesquisa tecnológica). A linguagem foi inspirada em OCaml e F# e foi projetada para o uso em programação, com foco em verificação formal de programas (Swamy et al., 2011).

A linguagem possui ampla adoção em verificação formal, sendo usada em diversos projetos, dentre os quais destacamos os seguintes: EverCrypt, uma biblioteca de criptografia verificada formalmente (Protzenko et al., 2020), e miTLS, uma implementação verificada do protocolo TLS (Bhargavan; Fournet; Kohlweiss, 2016).

Embora a linguagem tenha uma biblioteca padrão relativamente pequena, a linguagem possui forte integração com OCaml, podendo ser compilada para OCaml e permitindo o acesso a bibliotecas em OCaml. A linguagem também pode ser compilada para F# e C, e também há ferramentas externas que permitem a compilação para WebAssembly e assembly.

Para garantir que a verificação de tipos seja decidível, F* só permite que expressões comprovadamente terminantes sejam utilizadas em tipos dependentes, sendo necessário oferecer prova de terminação em situações em que o verificador de tipos não conseguir determinar a terminação.

3.4 Linguagens com tipos dependentes simulados

TypeScript

TypeScript é uma linguagem de programação criada com o objetivo de estender a linguagem JavaScript com um sistema de tipos mais robusto (Microsoft Corporation, 2024).

Embora a linguagem não implemente tipos dependentes, seu sistema de tipos possui vários recursos avançados que permitem simular tipos dependentes.

Um recurso de TypeScript que favorece a simulação de tipos dependentes são os tipos literais: cada literal possível da linguagem tem o seu próprio tipo *singleton* definido automaticamente. Como exemplo, a expressão `"abc"` é do tipo `"abc"`, que, por sua vez, é um subtipo de `string`. O mesmo ocorre com todos os valores possíveis dos tipos `number` e `boolean`. Embora a motivação principal dos tipos literais seja o seu uso na definição de tipos-união, é possível afirmar que os tipos literais representam valores dentro do sistema de tipos, permitindo a simulação de uma versão restrita de tipos dependentes.

Outros recursos da linguagem que permitem a construção de tipos complexos são: tipos-intersecção (que atuam efetivamente como a conjunção lógica entre restrições de tipo), tuplas e tipos recursivos. Serão fornecidos exemplos da simulação de tipos dependentes em TypeScript no Capítulo 4.

C

C possui um sistema de tipos restrito, uma vez que não implementa nenhuma forma de programação genérica, tampouco tipos dependentes. Contudo, é possível criar um macro do pré-processador da linguagem para simular tanto tipos de ordem superior quanto tipos dependentes, uma vez que ele atua antes mesmo do tempo de compilação e executa substituição simples de *strings*.

```
#define VECTOR(T, n, varName) \
struct {                      \
    T data[n];                \
    int length;               \
} varName = { .length = n }
VECTOR(int, 2 + 1, vec); //define uma variavel vec
```

Criado pelo autor deste trabalho, o macro acima cria uma variável no escopo local chamada `varName`, cujo tipo é a *struct* anônima definida no macro, que contém um array do tipo `T` com `n` elementos. O parâmetro `n` só aceita literais, operações entre literais e operações definidas em tempo de pré-processamento. O sistema de tipos não faz nenhuma validação posterior sobre a compatibilidade de tipos, permitindo que um objeto seja modificado com valores incompatíveis com a definição esperada.

C++

Com os templates de C++, é possível fazer uma simulação mais segura de tipos

dependentes em comparação com C.

```
template <typename T, int N> class Vector {
private:
    T data[N]; // Array interno para armazenar os elementos

public:
    T& operator[](int index) {
        return data[index];
    }

    constexpr size_t size() const {
        return N;
    }
};

Vector<int, 2 + 1> vec; //instancia o vetor
vec[0] = 322; //acessa o vetor
```

Criado pelo autor deste trabalho, o template de C++ define uma classe da linguagem de maneira mais automatizada e idiomática do que o macro em C. O modificador de acessibilidade garante que o array não será substituído por outro de tamanho incompatível. Contudo, não há verificação automática de limites do array, sendo necessário implementar no operador de indexação, sendo avaliado em tempo de execução. O parâmetro `int N` aceita qualquer expressão que seja `constexpr` (avaliável em tempo de compilação).

4 PROGRAMAÇÃO COM TIPOS DEPENDENTES

O tema central deste trabalho é a busca – e, se necessário, o desenvolvimento – de exemplos de programas e algoritmos que usem tipos dependentes. Os exemplos serão preferencialmente apresentados em Lean, com alguns exemplos em outras linguagens para explicitar suas diferenças e limitações.

4.1 Revisão sistemática

No desenvolvimento deste trabalho, a ferramenta Google Scholar foi muito utilizada para pesquisar na literatura referências sobre tipos dependentes e o seu uso em linguagens de programação. Inicialmente, encontramos uma vasta literatura sobre aspectos formais dos tipos dependentes. Contudo, a literatura sobre aspectos práticos de programação com tipos dependentes mostrou-se muito mais restrita.

Um aspecto relevante desta pesquisa é o fato de que muitos programadores não possuem o costume de publicar suas inovações em artigos acadêmicos, muitas vezes optando pelo uso de repositórios de código, páginas pessoais, páginas de documentação e enciclopédias *wiki*. Desta maneira, foi necessário expandir o escopo da busca inicial.

Analisando a literatura, encontramos artigos que descrevem as seguintes linguagens: Cayenne (Augustsson, 1998), uma extensão à linguagem ML incluindo uma implementação restrita de tipos dependentes (Xi; Pfenning, 1998), o sistema de tipos Deputy para C (Condit et al., 2007), Dependent ML (Xi, 2007) e seu sucessor ATS (Xi, 2017), Epigram (McBride, 2005) e F* (Swamy et al., 2011). Dessas linguagens, apenas F* teve adoção significativa, com todas as outras estando abandonadas ou tendo pouquíssima adoção. Utilizando o motor de busca Google, a ferramenta de pesquisa do repositório de código GitHub e as referências presentes em páginas na Wikipédia relacionadas a tipos dependentes, encontramos as seguintes páginas da web: a documentação oficial das linguagens Lean (Christiansen, David Thrane, 2023), Idris (Idris 2 Development Team, 2024), Agda (The Agda Team, 2024), F* (Swamy; Martínez; Rastogi, 2024).

Embora a proposta central deste trabalho seja a busca de trechos de código que utilizem tipos dependentes, encontramos poucos exemplos distintos. Na sequência deste capítulo, serão fornecidos os exemplos que encontramos e também alguns trechos de código que foram criados durante o desenvolvimento deste trabalho.

4.2 Vetor com comprimento fixo

O vetor com comprimento fixo (e definido no próprio tipo do vetor) é um dos exemplos mais comuns de uso de tipos dependentes em linguagens de programação, sendo parte da documentação oficial e da biblioteca padrão de diversas linguagens.

É importante frisar que, embora muitas linguagens de programação convencionais possuam algum mecanismo para lidar com vetores de tamanho fixo e conhecido, estes geralmente se resumem a algum tratamento em tempo de execução, sem a representação do comprimento dentro do sistema de tipos em si e, portanto, não se tratando de um exemplo de tipos dependentes.

Vetor com comprimento fixo - em Lean

Há mais de uma maneira de representar um vetor de comprimento fixo usando tipos dependentes. É possível reaproveitar o tipo `List` da linguagem, descrevendo de maneira explícita a restrição de comprimento com relação ao campo `length` da lista:

```
def VectList (n : Nat) (α : Type) := { l : List α // l.length = n }
def vec3 : VectList 3 Nat := ⟨[1, 2, 3], rfl⟩
```

Criada pelo autor deste trabalho, a definição de `VectList` acima utiliza a notação do tipo `Subtype`, que representa todos os valores de um tipo para os quais uma determinada proposição é verdadeira. O tipo `VectList` pode ser lido como "todos os elementos do tipo `List α` para os quais a propriedade `length` é igual a `n`.

A definição acima apresenta uma sintaxe mais simples comparada à definição indutiva que veremos a seguir e permite reaproveitar todas as funções que operam listas da biblioteca padrão. Contudo, o tipo apenas descreve a restrição de comprimento em alto nível e não oferece nenhuma representação estrutural do tipo, exigindo que seja fornecida uma prova de que a propriedade é válida cada vez que o tipo for instanciado. Um elemento do tipo `VectList` é um par ordenado que contém uma instância do tipo desejado e uma prova de que a propriedade é válida para aquela instância. No exemplo acima, a prova usada é `rfl`, que é a prova da reflexividade da igualdade entre números naturais, uma vez que o campo `length` de `[1, 2, 3]` avalia para `3` e `3 = 3`.

A definição de um vetor de comprimento fixo presente na documentação oficial de Lean é a seguinte:

```
inductive Vect (α : Type u) : Nat → Type u where
   | nil : Vect α 0
   | cons : α → Vect α n → Vect α (n + 1)
```

O tipo `Vect` é definido indutivamente, com um construtor para o caso base de comprimento 0 e outro construtor que define os demais comprimentos.

Alguns exemplos de definições de valores de algum dos tipos da família `Vect` são:

```
def vec0 : Vect String 0 := Vect.nil
def vec1 : Vect String 1 := Vect.cons "a" (Vect.nil)
def vec3 : Vect String 3 := vec1
```

As definições `vec0` e `vec1` são válidas. A definição `vec3` possui erro de tipo, cuja mensagem de erro apresentada pelo interpretador da linguagem é a seguinte:

```
type mismatch
  vec1
has type
  Vect String 1 : Type
but is expected to have type
  Vect String 3 : Type
Vect.cons "a" (Vect.nil)
```

No exemplo acima, a mensagem de erro identifica claramente o problema: a definição `vec3` espera um valor de tipo `Vect String 3` e recebe um valor do tipo `Vect String 1`. A mensagem de erro é facilmente compreendida em exemplos mais simples, porém veremos mais adiante que pode ser difícil o seu entendimento para casos mais complexos.

Como exemplo do uso de vetores de comprimento fixo, escolhemos implementar uma função que recebe dois vetores de comprimentos possivelmente distintos e retorna um novo vetor, com a primeira entrada concatenada com a segunda. A primeira tentativa feita pelo autor deste trabalho foi a seguinte:

```
def concat {a: Type u} {n m : Nat}
           (v1: Vect a n) (v2: Vect a m) : Vect a (n + m) :=
  match v1 with
  | .nil => v2
  | .cons x xs => .cons x (concat xs v2)
```

Embora a implementação esteja logicamente correta, ela esbarra em uma limitação do verificador de tipos do Lean:

```
type mismatch
  v2
has type
  Vect a m : Type u
but is expected to have type
  Vect a (0 + m) : Type u
```

O erro indica que o verificador de tipos não foi capaz de inferir que `0 + m = 0`. É necessário incluir a prova dessa igualdade, que vem da definição formal dos números naturais e já está implementada na biblioteca padrão da linguagem:

```
def concat {α: Type u} {n m : Nat}
           (v1: Vect α n) (v2: Vect α m) : Vect α (n + m) :=
  match v1 with
  | .nil => by rw [Nat.zero_add]; exact v2
  | .cons x xs => .cons x (concat xs v2)
```

A implementação acima apresenta o seguinte erro:

```
type mismatch
  Vect.cons x (concat xs v2)
has type
  Vect α (n✝ + m + 1) : Type u
but is expected to have type
  Vect α (n✝ + 1 + m) : Type u
```

O erro acima indica que o verificador de tipos não foi capaz de utilizar a comutatividade e associatividade da soma de naturais para inferir que `n + m + 1 = n + 1 + m`. É possível corrigir o erro reescrevendo a função da seguinte forma:

```
def concat {α : Type u} {n m : Nat}
    (v1 : Vect α n) (v2 : Vect α m) : Vect α (n + m) :=
  match n, v1 with
  | 0, .nil => by
      rw [Nat.zero_add]
      exact v2
  | _, .cons x xs => by
      rw [Nat.succ_add]
      exact Vect.cons x (concat xs v2)
```

O primeiro caso do *pattern matching* é o caso base, retornando o próprio `v2` caso o primeiro vetor seja vazio. É usada a prova `Nat.zero_add` para provar que `0+m = m`. O segundo caso constrói um novo vetor a partir do primeiro elemento de `v1` e chama a própria função recursivamente com o resto de `v1`. É utilizada a prova `Nat.succ_add`

para provar que `1 + (n-1) + m = n + m`.

As linguagens Idris, Agda e F também possuem uma implementação de vetor de comprimento fixo em sua biblioteca padrão.

Vetor com comprimento fixo - em Agda

A implementação da biblioteca padrão da linguagem Agda é a seguinte:

```
data Vector (A : Set) : Nat -> Set where
    []  : Vector A zero
    _::_ : {n : Nat} -> A -> Vector A n -> Vector A (suc n)
```

A biblioteca padrão da linguagem também implementa um operador infixo de concatenação de vetores ++:

```
_++_ : Vec A m → Vec A n → Vec A (m + n)
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Vetor com comprimento fixo - em Idris

A implementação da biblioteca padrão da da linguagem Idris é a seguinte:

```
data Vect : Nat -> Type -> Type where
    Nil  : Vect Z a
    (::) : a -> Vect k a -> Vect (S k) a
```

Na biblioteca padrão de Idris, `Z` corresponde ao número zero do tipo `Nat`. A biblioteca padrão da linguagem também implementa o operador infixo de concatenação de vetores ++:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

Vetor com comprimento fixo - em F*

A implementação da biblioteca padrão da linguagem F* é a seguinte:

```
type vec (a:Type) : nat -> Type =
  | Nil : vec a 0
  | Cons : #n:nat -> hd:a -> tl:vec a n -> vec a (n + 1)
```

A biblioteca padrão de F* implementa uma função `append` que concatena dois vetores:

```
let rec append #a #n #m (v1:vec a n) (v2:vec a m)
  : vec a (n + m)
  = match v1 with
    | Nil -> v2
    | Cons hd tl -> Cons hd (append tl v2)
```

Vetor com comprimento fixo - em TypeScript

Uma maneira de implementar vetor com comprimento fixo simulando tipos dependentes em TypeScript é a seguinte:

```
type Vector<T, N extends number> = readonly T[] & { length: N };
let arr4 : Vector<string, 4> = ["a", "b", "c", "d"];
```

No exemplo acima, criado pelo autor deste trabalho, o tipo `Vector<T, N>` é descrito como um array somente-leitura de `T` e que possua uma propriedade `length` do tipo `N`, que é o tipo literal associado ao comprimento desejado do vetor. Essa definição é semelhante à primeira definição em Lean que fornecemos. O seu uso é limitado e permite que `N` receba apenas literais ou constantes em tempo de compilação, não permitindo nenhum tipo de operação aritmética.

É possível fazer uma definição mais avançada, utilizando tipos recursivos (semelhante a tipos indutivos) combinados com tipos-intersecção e tuplas do TypeScript para definir a estrutura exata do tipo desejado:

```
type VecBuilder<T, N extends number, Acc extends T[] = []> =
  Acc['length'] extends N
    ? Acc
    : VecBuilder<T, N, [...Acc, T]>;

type Vec<T, N extends number> = VecBuilder<T, N>;
```

A definição acima, também criada pelo autor deste trabalho, é semelhante à definição indutiva que vimos nas linguagens Lean e Agda. `Acc` serve como um acumulador dentro do sistema de tipos, definido inicialmente como o array de comprimento 0 e sendo expandido a cada chamada recursiva do tipo até o tamanho definido por `N`. Como exemplo, o tipo `Vec<string, 3>` é avaliado pelo verificador de tipos e expandido para a

tupla `[string, string, string]`.

A principal vantagem da definição recursiva é que, embora TypeScript não permita operações aritméticas entre tipos literais, é possível usar a própria definição de `Vec` para criar um "tipo gerador de tipos"que efetue as operações. Um exemplo de operação entre tipos literais é o seguinte:

```
type Add<N extends number, M extends number> =
  Extract<[...Vec<null, N>, ...Vec<null, M>]['length'], number>;

const vec : Vec<string, Add<1, 2>> = ["a", "b", "c"];
```

Criado pelo autor deste trabalho, o tipo `Add<N,M>` usa a própria definição de `Vec` para construir vetores de comprimento `N` e `M`, concatenar o "tipo expandido"de cada um deles e obter o tipo literal correspondente à propriedade `length` desse tipo expandido, efetivamente calculando `N + M` dentro do sistema de tipos.

4.3 Verificação de limites de vetor

A verificação de limites de vetor é um problema que pode ser expresso de maneira simples usando tipos dependentes. Embora possa parecer um exemplo simplório, foi uma das primeiras motivações para propostas de linguagens com tipos dependentes, como vimos em Xi and Pfenning (1998).

Verificação de limites de vetor - em Lean

A biblioteca padrão de Lean possui uma definição do tipo `Fin n`, que representa o tipo de qualquer número natural menor que `n`. Contudo, esse tipo implementa aritmética modular, aceitando valores maiores que `n`. Para a verificação de limites em acesso a vetor, é necessário definir um tipo que aceite apenas naturais que sejam menores que `n`, sendo uma expressão mal-tipada em outros casos:

```
structure Finite (n : Nat) where
    val  : Nat
    isLt : LT.lt val n
```

Criado pelo autor deste trabalho de maneira semelhante ao tipo `Fin n` da biblioteca padrão de Lean, o tipo `Finite n` é definido como um número natural `val` e uma prova `isLt` de que `val` é menor do que `n`. A partir dessa definição, e utilizando a defi-

nição indutiva de um vetor de tamanho fixo da seção 4.2, é possível definir uma função que acesse uma determinada posição do vetor, com a garantia em tempo de compilação de que o índice está dentro dos limites do vetor:

```
def getAt {α : Type u} {n : Nat} (v : Vect α n) (i : Finite n) : α :=
  match v, i with
  | Vect.cons x _, ⟨0, _⟩ => x
  | Vect.cons _ xs, ⟨i+1, h⟩ => getAt xs ⟨i, Nat.lt_of_succ_lt_succ h⟩

def vec3 := Vect.cons 9 (Vect.cons 8 (Vect.cons 7 (Vect.nil)))

#eval getAt vec3 ⟨2, by decide⟩ --avalia para 7
#eval getAt vec3 ⟨3, by decide⟩ --erro de tipo
```

No exemplo acima, criado pelo autor deste trabalho, a primeira avaliação retorna o valor presente no índice 2 do vetor. A segunda avaliação apresenta o seguinte erro de tipo:

```
tactic 'decide' proved that the proposition
  3 < 0 + 1 + 1 + 1
is false
```

A tática `decide` é útil para gerar automaticamente uma prova de proposições simples, que não precisam de indução ou substituições complexas. Na segunda avaliação, ela consegue provar que `3 < 3` é falso e, portanto, a expressão é mal-tipada.

4.4 Matriz com dimensões fixas

Uma aplicação que se beneficia de tipos dependentes é o uso e manipulação de matrizes. Várias operações matemáticas envolvendo matrizes são válidas a depender das dimensões das matrizes de entrada, com o exemplo mais famoso sendo a multiplicação: dada uma matriz `A` de dimensões `m × n` e uma matriz `B` de dimensões `n x p`, a dimensão da matriz de saída é `m × p`. É importante notar que `n` é tanto o número de colunas de `A` quanto o número de linhas de `B`, com o produto sendo indefinido para demais casos.

Na maioria das linguagens de programação, a multiplicação de matrizes é comumente implementada com a validação da compatibilidade das dimensões sendo feita manualmente em tempo de execução. A implementação usando tipos dependentes, ao codificar as dimensões dentro do sistema de tipos, permite que os erros mais comuns sejam detectados em tempo de compilação.

É possível definir uma matriz de dimensões fixas baseada na definição de `Vect` do exemplo anterior. Nesta seção, todas as definições são de autoria do criador deste trabalho.

```
def Matrix (α : Type u) (m n : Nat) := Vect (Vect α n) m
```

O tipo `Matrix α m n` é definido como um vetor de vetores, de maneira semelhante a outras linguagens de programação.

Para definirmos uma função que implemente a multiplicação de matrizes com as dimensões fixas no sistema de tipos, é conveniente definirmos algumas funções auxiliares:

```
-- definição simplificada, com o tipo Nat fixo
def MatrixNat (m n : Nat) := Matrix Nat m n

-- aplica a função f em cada elemento do vetor
def map {α β : Type u} (f : α → β) : {n : Nat} → Vect α n → Vect β n
  | _, .nil        => .nil
  | _, .cons x xs  => .cons (f x) (map f xs)

-- zip com map
def zipWith {α β γ : Type u} (f : α → β → γ)
    : {n : Nat} → Vect α n → Vect β n → Vect γ n
  | _, .nil, .nil               => .nil
  | _, .cons x xs, .cons y ys   => .cons (f x y) (zipWith f xs ys)

-- cria um vetor com n cópias de x
def replicate {α : Type u} (x : α) : (n : Nat) → Vect α n
  | 0     => .nil
  | n+1   => .cons x (replicate x n)

-- cria a matriz transposta
def transpose {m n : Nat} : MatrixNat m n → MatrixNat n m
  | .nil => replicate .nil n
  | .cons row rows =>
    zipWith Vect.cons row (transpose rows)
```

Utilizando o tipo `MatrixNat` e as funções auxiliares definidas acima, é possível implementar o produto interno de dois vetores e, a partir dele, a multiplicação de matrizes:

```
-- produto interno
def dot {n : Nat} (xs ys : VectNat n) : Nat :=
  match xs, ys with
  | .nil, .nil => 0
```

```
  | .cons x xs, .cons y ys =>
      x*y + dot xs ys

def matrixMul {m n p : Nat}
    (A : MatrixNat m n) (B : MatrixNat n p) : MatrixNat m p :=
  let bt := transpose B
  map (fun (rowA : VectNat n) => map (dot rowA) bt) A
```

A definição acima utiliza o produto interno entre as linhas de `A` e as colunas de `B` (acessados a partir de sua matriz transposta) para calcular o produto `A × B`.

4.5 Árvore binária completa

Em Bove and Dybjer (2008), foi encontrado um exemplo de árvore binária completa em Agda, utilizando um número natural na definição do tipo para garantir que todas as subárvores de mesmo nível possuem a mesma altura, garantindo que todas as folhas estão no último nível da árvore.

```
data DBTree (A : Set) : Nat -> Set where
    dlf : A -> DBTree A zero
    dnd : {n : Nat} -> DBTree A n -> DBTree A n -> DBTree A (succ n)
```

O construtor `dlf` define uma folha como uma `DBTree` de altura 0. O construtor `dnd` define os demais nós da seguinte maneira: se duas subárvores possuem altura `n`, o nó superior possui altura `succ n`.

É possível fazer definição equivalente em Lean:

```
inductive DBTree (A : Type) : Nat → Type
    | dlf : A → DBTree A 0
    | dnd {n : Nat} : DBTree A n → DBTree A n → DBTree A (n + 1)
```

4.6 Considerações finais

Neste Capítulo, foram apresentados alguns exemplos de código que usam tipos dependentes. Alguns exemplos foram obtidos na documentação e nas bibliotecas-padrão das linguagens, enquanto outros foram desenvolvidos pelo autor deste trabalho. Ao longo do desenvolvimento do trabalho, nos deparamos com uma escassez de exemplos de uso de tipos dependentes, com as documentações das linguagens fornecendo apenas exemplos

idênticos uns aos outros ou apresentando exemplos artificiais e sem aplicação prática.

5 CONCLUSÃO

Este trabalho foi iniciado por uma definição informal de tipos dependentes, avançando para uma descrição do isomorfismo de Curry-Howard e sua relação com tipos dependentes, a importância dos tipos dependentes para a prova de teoremas, as vantagens do uso de tipos dependentes na programação e o impacto de tipos dependentes na verificação de tipos. Na sequência, foi dada uma visão histórica e comparativa de linguagens de programação com tipos dependentes, detalhando suas características e limitações. Por fim, fornecemos alguns exemplos de programas e funções que utilizam tipos dependentes para expressar e garantir propriedades desejadas.

Ao longo do desenvolvimento deste trabalho, foi possível observar que o uso de tipos dependentes em programação está atualmente restrito a linguagens associadas a provadores de teoremas ou linguagens criadas com foco em verificação formal, sendo virtualmente inexistente em linguagens *mainstream*. Essa baixa disponibilidade prejudica a adoção e o conhecimento de tipos dependentes por parte dos programadores.

Há diversos outros fatores que contribuem para o pouco uso de tipos dependentes: dificuldade de uso, muitas vezes exigindo provas formais até para casos relativamente simples; sintaxe avançada, se assemelhando a formalismos lógicos; mensagens de erros complexas, dificultando o entendimento dos erros no desenvolvimento. Atualmente, todas essas dificuldades limitam os tipos dependentes às áreas de prova de teoremas e verificação formal de programas.

5.1 Trabalhos futuros

Com o objetivo de facilitar a adoção de tipos dependentes, um trabalho futuro interessante seria a escolha de uma linguagem *mainstream* (como Rust ou C#) para uma proposta de extensão da linguagem, implementando uma versão restrita de tipos dependentes.

REFERÊNCIAS

AUGUSTSSON, L. Cayenne—a language with dependent types. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 34, n. 1, p. 239–250, 1998.

BHARGAVAN, K.; FOURNET, C.; KOHLWEISS, M. mitls: Verifying protocol implementations against real-world attacks. **IEEE Security & Privacy**, IEEE, v. 14, n. 6, p. 18–25, 2016.

BOVE, A.; DYBJER, P. Dependent types at work. In: **International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development**. [S.l.]: Springer, 2008. p. 57–99.

BRADY, E. **Type-driven development with Idris**. [S.l.]: Simon and Schuster, 2017.

BRUIJN, N. G. D. The mathematical language automath, its usage, and some of its extensions. In: **Studies in Logic and the Foundations of Mathematics**. [S.l.]: Elsevier, 1994. v. 133, p. 73–100.

Christiansen, David Thrane. **Functional Programming in Lean**. 2023. <https://lean-lang.org/functional_programming_in_lean/>. Acesso em: 17 dez. 2024.

CONDIT, J. et al. Dependent types for low-level programming. **Programming Languages and Systems: 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 16**, p. 520–535, 2007.

CONSTABLE, R. L. et al. **Implementing Mathematics with the Nuprl Proof Development System**. 1985. <https://nuprl-web.cs.cornell.edu/book/>. Acesso em: 18 nov. 2024.

COQUAND, T.; HUET, G. **The calculus of constructions**. Thesis (PhD) — INRIA, 1986.

CURRY, H. B. Functionality in combinatory logic. **Proceedings of the National Academy of Sciences**, National Acad Sciences, v. 20, n. 11, p. 584–590, 1934.

GONTHIER, G. et al. Formal proof–the four-color theorem. **Notices of the AMS**, v. 55, n. 11, p. 1382–1393, 2008.

HOWARD, W. A. The formulae-as-types notion of construction. **To HB Curry: essays on combinatory logic, lambda calculus and formalism**, v. 44, p. 479–490, 1980.

Idris 2 Development Team. **A Crash Course in Idris 2**. 2024. <https://idris2.readthedocs.io/en/stable/tutorial/>. Acesso em: 19 dez. 2024.

LEROY, X. et al. Compcert-a formally verified optimizing compiler. In: **ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress**. [S.l.: s.n.], 2016.

MARTIN-LÖF, P. An intuitionistic theory of types: predicative part. **Studies in Logic and the Foundations of Mathematics**, Elsevier, v. 80, p. 73–118, 1975.

MCBRIDE, C. Epigram: Practical programming with dependent types. In: SPRINGER. **Advanced Functional Programming: 5th International School, AFP 2004, Tartu, Estonia, August 14–21, 2004, Revised Lectures**. [S.l.], 2005. p. 130–170.

Microsoft Corporation. **The TypeScript Handbook**. [S.l.], 2024. Acesso em: 18 dez. 2024. Available from Internet: <https://www.typescriptlang.org/docs/handbook/intro.html>.

NORELL, U. **Towards a practical programming language based on dependent type theory**. [S.l.]: Chalmers University of Technology, 2007.

PIERCE, B. C. **Types and programming languages**. [S.l.]: MIT press, 2002.

PROTZENKO, J. et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In: IEEE. **2020 IEEE Symposium on Security and Privacy (SP)**. [S.l.], 2020. p. 983–1002.

SWAMY, N. et al. Secure distributed programming with value-dependent types. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 46, n. 9, p. 266–278, 2011.

SWAMY, N.; MARTíNEZ, G.; RASTOGI, A. **Proof-Oriented Programming in F***. 2024. <https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf>. Acesso em: 19 dez. 2024.

The Agda Team. **Agda User Manual - Release 2.8.0**. 2024. <https://agda.readthedocs.io/en/latest/overview.html>. Acesso em: 19 dez. 2024.

The Coq Development Team. **The Coq Reference Manual – Release 8.20.0**. 2024. <https://coq.inria.fr/doc/V8.20.0/refman>. Acesso em: 12 dez. 2024.

XI, H. Dependent ml an approach to practical programming with dependent types. **Journal of Functional Programming**, Cambridge University Press, v. 17, n. 2, p. 215–286, 2007.

XI, H. Applied type system: An approach to practical programming with theorem-proving. **arXiv preprint arXiv:1703.08683**, 2017.

XI, H.; PFENNING, F. Eliminating array bound checking through dependent types. **Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation**, p. 249–257, 1998.