Introdução a sistemas de tipos e tipos algébricos

G. L. Conegero[1]

[1]Departamento de Informática, Curitiba-PR
Universidade Federal do Paraná

September 9, 2026

Índice

Introdução: Sistema de tipos

- O que são sistemas de tipos e por que estudá-los?
- Análise semântica.
  - Tabela de símbolos.
  - Tipos de dados.
- Linguagens funcionais[1]

[1] Haskell, OCaml, ML, Elm, etc.

Introdução: Definição(ões)

Definição 1: Um sistema de tipos é um método sintático tratável para provar a ausência de certos comportamentos de programa, classificando frases de acordo com os tipos de valores que elas computam [Pierce 2002].
Definição 2: Tipos são o princípio organizador central da teoria de linguagens de programação. Os recursos da linguagem são manifestações da estrutura de tipos. A sintaxe de uma linguagem é governada pelos construtos que definem seus tipos, e sua semântica é determinada pelas interações entre esses construtos. A solidez[2] do design de uma linguagem - a ausência de programas mal definidos - segue naturalmente [Harper 2012].
Definição 3: Um sistema de tipos é um conjunto de regras que atribui uma propriedade, conhecida como um tipo, aos construtos de um programa, como variáveis, expressões e funções. O propósito primário é reduzir bugs prevenindo operações que não fazem sentido, como dividir um número por uma string. [Diehl 2024].

[2] Original: *soundness*.

Introdução: Definição

Um sistema de tipos é:

- Regras semânticas.
- Ausência de comportamento indesejável.
- Uma teoria sobre linguagens de programação.
- Lógica*.

Introdução: IA...

- 25% do código na Google é gerado por IA.
- Navier-Stokes foi resolvido com IA[3]
- Linux Foundation tem política para uso de IA.
- Vários por cento de vocês usam IA para gerar código (trabalho, faculdade, etc.).

Mas como confiar que a implementação está correta? Testes?

Utilizando alguma lógica que permita falar sobre a linguagem. Mas isso já existe, o sistema de tipos!!!!

[3] LEIAM: `https://cims.nyu.edu/~tristanb/statement.pdf`

Exemplo: Linguagem C

A linguagem C possui tipos, e formas de criar novos tipos.

- Tipos intrinsecos: `int`, `float`, `char`, `*void`, `*int`, `char[34]`.
- Tipos derivados: `structs`, `unions`, `enum`.
- Tipos de funcão: `int func(char *str, struct TipoA)`
- Proibe a operação em tipos diferentes: int + char*, `double - *int`, etc.

Bom para abstrair o conceito de memória como uma tabela de valores.

Exemplo: Linguagem C

Como ter uma abstração de alto nível, para os seguintes conceitos:

- Métodos de um tipo. Quais operações podem operar aquele tipo.
- Fluxo de erros/*side effects*.
- Segurança da memória.
- Concorrência e paralelismo.

Exemplo: Java

Novo conceito: Exceções ou *Effects*. Sistema de exceções não é sistema de tipos, mas ambos podem se complementar.

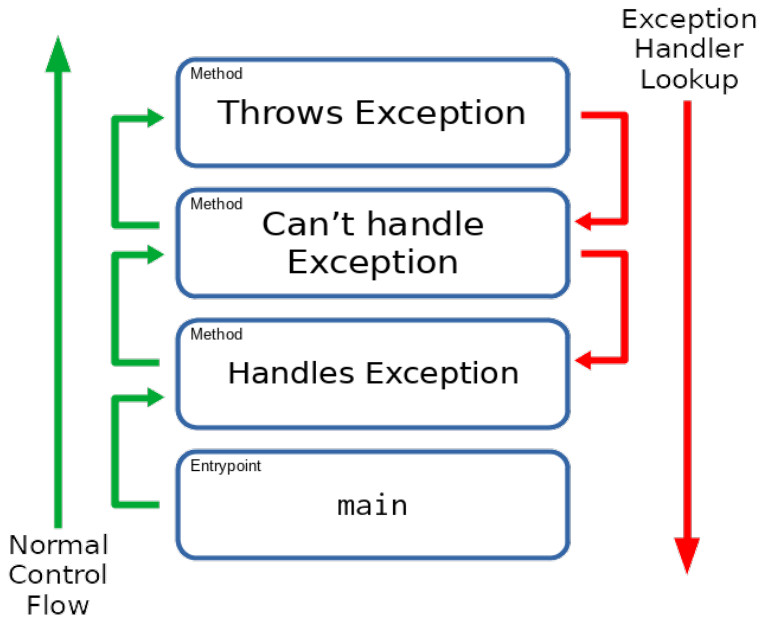

Figure: Sistema de exceções em Java

Limitações da linguagem C

Considere um sistema que utiliza a seguinte `struct`.

```
struct User {
    char *name;
    struct {
        char *email;
        char *telefone;
    } contato;
};
```

Limitações da linguagem C

Como voce implementaria a seguinte função:

```
void enviaNotificacao(struct User u, char *msg) {
    // Deve usar o contato do usuario para enviar
        notificacao
}
```

Limitações da linguagem C

Uma alternativa de implementação:

```
void enviaNotificacao(struct User u, char *msg) {
    // Deve usar o contato do usuario para enviar
        notificacao
    if (u.contato.email != NULL) {
        sendEmail(u.contato.email, msg);
        return;
    }

    if (u.contato.telefone != NULL) {
        sendSMS(u.contato.telefone, msg);
        return;
    }

    // ERRO, nao deveria chegar aqui nunca
    // Como tratar?
    return;
}
```

Limitações da linguagem C

Primeiro passo, utilizar um tipo melhor para representar strings. Vamos considerar que uma valor do tipo `string` nunca vai ser NULL.

```
// Primeiro, vamos ter um tipo que representa uma string de
    verdade, nao pode ser null.
struct User {
    string name;
    struct {
        string email;
        string telefone;
    } contato;
};
```

Limitações da linguagem C

Porém, nessa representação o tipo diz que sempre vamos ter email e telefone. O que podemos fazer? Usar o tipo algébrico `optional`.

```
struct User {
    string name;
    struct {
        optional<string> email;
        optional<string> telefone;
    } contato;
};
```

Limitações da linguagem C

Demos uma volta!!! Então como resolver? Mais tipos algébricos. Usando `variant`'s/*Sum Types*.

```
struct User {
    string name;
    variant {
        Email(string);
        Telefone(string);
        Both(string, string);
    } contanto;
};
```

Esse, em específico, podemos chamar de *Tagged Unions*.

Limitações da linguagem C

Agora sim, podemos ter uma implementação:

```
// Agora, nao tem como criar um valor do tipo struct User
    sem
// que ele tenha pelo menos email ou telefone validos.
void enviaNotificacao(struct User u, char *msg) {
    switch(u.contato) {
        case Email(email): sendEmail(email, msg);
        case Telefone(tel): sendSMS(tel, msg);
        case Both(both): sendEmail(both[0], msg);
    }

    // Garantido que pelo menos um dos ramos do switch vai
    // acontecer. Compilador sabe disso. Portanto
    // aqui e garantido que tudo deu certo
}
```

Limitações da linguagem C

Tem como eu simular essa implementação em C puro.

1. Usar `enum` com os campos `Email, Telefone e Both`.
2. Usar `union` com todos os campos.
3. Ativar a flag de compilação *-Wswitch-enum*.

Não garante o certificado para o compilador.

Limitações da linguagem C

```
// Compilar com -Wswitch-enum
enum ContatoTipo {
    Email,
    Telefone,
    Both,
};

struct User {
    string name;
    enum ContatoTipo tipo_contato;
    union {
        string email;
        string telefone;
        struct {
            string email;
            string telefone;
        } both;
    } contato;
}
```

Tipos algébricos

O que são tipos algébricos?

[4] Especificamente `unions` em conjunto com um campo de especificação, um `enum`.

Tipos algébricos

O que são tipos algébricos?
Tipos algébricos são uma forma de construir novos tipos.

[4] Especificamente `unions` em conjunto com um campo de especificação, um `enum`.

Tipos algébricos

O que são tipos algébricos?
Tipos algébricos são uma forma de construir novos tipos.
Na verdade, nós já os conhecemos. São as `structs` e `unions`[4] na linguagem C.

[4] Especificamente `unions` em conjunto com um campo de especificação, um `enum`.

Tipos algébricos: Produto

Um tipo produto agrupa 2 ou mais tipos em um tipo só. Existem duas variantes, tuplas e `records/structs`.

```
// Produto entre int e int
// Tupla
type Point = (int, int);
// Acesso por posicao da tupla
Point v1 = (5, 6);
v1[0] == 5
v1[1] == 6
// Record
type Vector = record { int x, int y};
// Acesso pelo nome
Vector v2 = {x = 5, y = 6};
v2.x == 5
v2.y == 6
```

Tipos algébricos: Soma

Usado para abstrair a ideia de escolha ao nível de tipos. Onde um tipo pode ser uma das opções listadas.

```
// Soma entre int e string
type Contato = variant {
    Email string;
    Telefone string;
    Both (string, string);
};
// Cria  o de valor por "injecao"
Contato v = Email("email@email.com");
// Acesso do valor por switch case
switch (v) {
    case Email(email): ...
    case Telefone(tel): ...
    case Both(both): ...
}
```

Tipos algébricos: tipo *Optional*

Tipo que captura a possibilidade da não existência de um valor. O verdadeiro tipo de ponteiros em C.

```
type PonteiroInt = variant {
    // Teoricamente, NULL aplica sobre o tipo unit
    NULL;
    PTR int;
};

// Vamos usar polimorfismo/genericos.
// Assunto para outra aula
type optional<a> = variant {
    Nothing;
    Some a;
};
```

Tipos algébricos: Teoria

- Quantos valores tem para bool?
- Quantos valores tem para `int`?
- Tamanho do tipo: $|\texttt{int}| = 2^{32}$, $|\texttt{bool}| = 2$.
- Produto: multiplicação entre tamanhos.
  $|(\texttt{int}, \texttt{bool})| = |\texttt{int}| * |\texttt{bool}| = 2^{32} * 2 = 2^{33}$
- Soma: adição entre tamanhos.
  $|\texttt{variant } \{\texttt{A int, B bool}\}| = |\texttt{int}| + |\texttt{bool}| = 2^{32} + 2$

Tipos algébricos: Teoria

- Estamos falando de álgebra, precisamos das constantes 0 e 1.
- Tipos especiais `unit` e `void`.
- $|\texttt{unit}| = 1$ e $|\texttt{void}| = 0$.

Tipos algébricos: Prova de ausência[5]

Considere nosso tipo contato no exemplo anterior.

```
// Alternativa 1
type Contato = record {
    optional<A> email;
    optional<B> telefone;
};

// Alternativa 2
type Contato = variant {
    Email A;
    Telefone B;
    Both (A, B);
};
```

[5] Exemplo retirado de https://tomasp.net/blog/types-and-math.aspx/

Tipos algébricos: Prova de ausência

Tamanho `optional<int>`:

$$|\texttt{optional<int>}| = 1 + |\texttt{int}|$$

Quais valores são eles:

- `Nothing`
- `Some 0`
- `Some 1`
- `Some 2`
- ...
- `Some` $2^{32} - 1$

Tipos algébricos: Prova de ausência

Tamanho alternativa 1:

$$
\begin{aligned}
|\texttt{optional<X>}| &= 1 + |\texttt{X}| \\
|\texttt{Contato}| &= (1 + |\texttt{A}|) * (1 + |\texttt{B}|) \\
&= 1 * 1 + 1 * |\texttt{B}| + |\texttt{A}| * 1 + |\texttt{A}| * |\texttt{B}| \\
&= |\texttt{A}| * |\texttt{B}| + |\texttt{A}| + |\texttt{B}| + 1
\end{aligned}
$$

Tamanho alternativa 2:

$$
|\texttt{Contato}| = |\texttt{A}| + |\texttt{B}| + (|\texttt{A}| * |\texttt{B}|)
$$

Tipos algébricos: Prova de ausência

$$|\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}| + 1 \neq |\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}|$$

Tipos algébricos: Prova de ausência

$$|\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}| + 1 \neq |\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}|$$

Definição 1 de sistemas de tipos.

Tipos algébricos: Prova de ausência

$$|\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}| + 1 \neq |\mathtt{A}| * |\mathtt{B}| + |\mathtt{A}| + |\mathtt{B}|$$

Definição 1 de sistemas de tipos.
Prova da ausência do estado $(\mathtt{NULL}, \mathtt{NULL}) \sim 1 * 1$.

Conclusão

- Sistemas de tipos são importantes.
- A teoria está no nosso dia a dia.
- Tipos algébricos são apenas a ponta do iceberg.

Conclusão

- Polimorfismo de tipos.
- Inferência de tipos.
- Verificação bidirecional.
- Verificação dinâmica ou estática.
- Tipos lineares.
- Tipos dependentes (*Dependent Types*)
- Muito, muito mais.

Linguagens interessantes

Linguagens com sistemas de tipos interessantes.

- Haskell: GADTS, and more
- ML: Hindley-Milner
- Rust: Memory Safety
- Typescript: ...
- Prism: Effects
- Roc: Side Effects
- F#: Best functional language
- Elixir: Concurrency (Brasil)
- Idris: Dependent Types
- Lean: AI proof tool (Brasil)

Referências I


Diehl, S. (2024).
Typechecker zoo.
Disponível em: https://sdiehl.github.io/typechecker-zoo/introduction.html.
Acesso em: 07 set. 2026.


Harper, R. (2012).
Practical Foundations for Programming Languages.
Cambridge University Press, Cambridge, UK.


Pierce, B. C. (2002).
Types and Programming Languages.
MIT Press, Cambridge, MA, USA.