Um resumo do Artigo *“On Understanding Types, Data Abstraction, and Polymorphism”*

Gustavo Lopes, Lucas Santiago, Thiago Henriques

1 - Objetivo

O artigo em questão faz uma visita à história de tipagem pelas linguagens de programação, e os conhecimentos adquiridos temporalmente sobre a teoria de tipagem, abstração de dados e polimorfismo. Com isso, Luca Cardelli e Peter Wegner fazem a modelagem de uma linguagem de programação(batizada de FUN), que faz um melhor aproveitamento de fortes técnicas para a programação visual de fortes linguagens orientadas a objetos e para sistemas de tipos que sejam mais expressivos e poderosos.

1.1 Representação de dados

Para começar o artigo, o autor em vez de ir pela definição diretamente, ele dá exemplos de linguagens sem tipagem e o processo de como a tipagem foi naturalmente sendo incorporado.

Não ter tipagem significa ter uma tipagem só. Na memória do computador isso é representado por “bit strings”. Em LISP, são as S-expressions. No Cálculo lambda, são as expressões lambda. Por fim, também tem os Sets em Set Theory.

A ideia de possuir tipos foi naturalmente implementada, com a necessidade de possuir diferentes formatos com diferentes usos e distintos comportamentos. Além disso, é muito importante proteger a variável de ser interpretada de forma incorreta. O uso, por exemplo, de um inteiro com um ponteiro poderia ser desastroso ao se acessar arbitrariamente os dados de outro programa em execução.

Porém, ainda é muito difícil fazer uma distinção completa entre a organização desses tipos e de fazer realmente uma linguagem com tipagem. Um exemplo disso seria ter uma função lambda que retorna booleano ou inteiro.

1.2 Tipagem forte e fraca

Uma maneira de descrever tipos é comparado com uma armadura. Armaduras protegem o usuário de danos exteriores. Em tipagem, a armadura protege os dados de serem usados de forma não desejadas ou não intencionadas. Isso acontece, pois, como objetos possuem certas tipagem, ele precisa seguir as, funcionalidade da tal tipagem.

O problema é que mesmo assim, durante a compilação, pode acontecer de essa regra não ser obedecida e causar muitos problemas. Uma solução para esse problema é a tipagem estática.

Tipagem estática significa que variáveis de um programa são definidas explicitamente e então checadas durante tempo de compilação. Isso é importante para a checagem de erros, melhoria no desempenho, garante uma estrutura a ser respeitada e de fácil leitura. Porém, existe outra categoria de tipagem, que garante maior flexibilidade, a tipagem forte.

Tipagem forte são linguagens onde cada categoria de dado, são predefinidos como parte da linguagem

1.3 Tipos de polimorfismo

Linguagens com tipagem convencional, não suportam que uma mesma variável tenha mais de um tipo e são chamadas de monomórficas, enquanto linguagens polimórficas possuem a ideia contrária: valores e variáveis talvez tenham mais de um tipo e suas operações são aplicáveis a operandos de mais de um tipo.

Existem dois grandes grupos de polimorfismo: universal e não universal(Ad-hoc).

1.3.1 Universal

Polimorfismo universal assume que um tipo pode assumir um infinito número de diferentes tipos.

Eis os subtipos do polimorfismo universal:

- Polimorfismo paramétrico: permitir uma função ou um tipo de dados funcionar com uma gama de tipos diferentes, exibindo algum tipo de estrutura em comum(genérico). Ex: templates em C++, ArrayList em Java.

- Polimorfismo de inclusão: habilidade de classificar subtipos utilizando herança. Ex: *Extends* em Java

1.3.2 Não-universal(Ad-hoc)

Ad-hoc é uma categoria de polimorfismo no qual objetos podem ser aplicados a argumentos de diferentes tipos, já que objetos polimórficos podem denotar uma série de implementações distintas e potencialmente heterogêneas, dependendo dos argumentos ao qual é aplicada.

Subtipos de polimorfismo de Ad-hoc incluem:

- Polimorfismo de sobrecarga: uma estrutura funciona em diferentes tipos, e podem funcionar de maneira diferente para cada tipo. Também chamado de Overloading, este processo é puramente uma maneira sintática para usar nomes iguais para objetos diferentes. Em tempo de compilação, o compilador pode resolver a ambiguidade. Ex: Funções com o mesmo nome, mas que recebem parâmetros com tipos diferentes.

- Polimorfismo de coerção: operação semântica necessária para converter um tipo para outro. Isso é necessário para evitar erros de compilação e às vezes o próprio compilador já faz isso. Ex: soma de inteiros com números reais

A distinção de sobrecarga e coerção pode ser um pouco confusa em algumas situações, principalmente em línguas sem tipagem e tipagem, ou até mesmo em estáticas e compiladas.

1.3.3 Polimorfismo em monomorfismo

Estas definições de polimorfismo apresentadas pelo autor são aplicáveis apenas em linguagens com uma clara noção de tipo e valor. Isso se torna um problema, pois linguagens estritamente monomórficas seriam muito restritivas em seu poder de expressão. Algumas linguagens conseguem tirar algumas das restrições impostas pelo padrão do monomorfismo, como, por exemplo: Ada e Pascal. Estas línguas estendem o conceito de monomorfismo, aplicando as regras do polimorfismo através das maneiras que elas conseguem.

- Overloading(Sobrecarga)
- Coercion(coerção)
- Subtyping(inclusão)
- Value Sharing(paramétrico)

Lembrando apenas de que: por mais que existam paradigmas do polimorfismo presente, nenhuma dessas linguagens são "verdadeiramente polimórficas".

Para ser considerado polimorfismo verdadeiro, é preciso que um código consiga rodar várias estruturas e tipos com apenas uma função, mesmo que em alguns casos seja possível criar várias funções para cada um dos tipos e estruturas em tempo de compilação para gerar código de máquina mais otimizado.

1.4 A evolução de tipos em linguagens de programação

Essa história começa com a linguagem de programação FORTRAN. Essa linguagem tinha uma propriedade interessante: diferenciar inteiros de números reais. O motivo pelo qual essa distinção era feita, é devido à necessidade de distinguir os inteiros para a realização de iteração em loops e computação em arrays.

Outras linguagens também pegaram carona nessa ideia, linguagens como o ALGOL 60 tinham também essa distinção, porém a distinção era entre inteiros, reais e

booleanos. O ALGOL 60 foi a primeira linguagem a ter a explícita noção de tipos e requerimentos associados em tempo de compilação para verificação de tipos.

Com o decorrer do tempo, as linguagens de programação foram aumentando a quantidade de tipos presentes: arrays, ponteiros, strings, char, bytes e arquivos são apenas alguns exemplos desses tipos que foram adicionados.

Simula foi a primeira linguagem orientada a objetos. As variáveis que guardavam as instâncias das classes não eram excluídas após a execução dos seus procedimentos. Além disso, suas estruturas e interfaces eram visíveis para os usuários. ADA por sua vez, tinha uma rica variedade de módulos, incluindo subprogramas, abstração de dados, e funções para suportar programação concorrente.

1.5 Sub língua de expressão de tipos

Um dos objetivos deste artigo é examinar os aspectos positivos e negativos entre riqueza e tratabilidade(acessibilidade para checagem de tipo durante compilação) para sub-línguas de expressão de tipos. Basicamente, existe todo um vocabulário de palavras usadas para definir os tipos em linguagens de programação, sub linguagens de expressão de tipos incluem as categorias de dados básicos, como: inteiros e booleanos.

Sub-línguas de expressão de tipos precisam ser tanto para denotar tipo, mas também para mostrar equivalência/similaridade a quais tipos. O objetivo é que a relação entre tipos possam ser expressas e computáveis. Similaridade entre tipos pode ser referido como polimorfismo.

1.6 Cálculo Lambda

Cálculo lambda foram criados para que uma mesma variável ou função suporte vários tipos, ou estruturas. Dessa forma, tal estrutura pode ser usada como uma variável, permitindo que esta seja executada por si só. Por exemplo: função(função(y)). Essa forma de equação consegue operar sobre diferentes tipos e estruturas. Com isso, é importante ter alguma forma de checagem em compilação ou tempo de execução, visto que a operação de divisão, por exemplo, não faria sentido algum em um contexto de uma árvore binária.

Para resolver o problema citado anteriormente, a ideia de um cálculo lambda tipado ganhou força. Para colocar tipo nessas operações, o compilador precisa fazer checagem de valores para eliminar a possibilidade de uma equação impossível, tanto quanto gerar um erro para o programador.

Outra forma de evitar erros no código é - como a linguagem FUN criada pelos autores do artigo - permitir que quando uma função-λ for criada, que ela suporte separadamente cada tipo em “sub-funções” e a partir da entrada da função faça uma escolha a partir do tipo. Essa decisão evita que operações indevidas aconteçam com o código, operações não permitidas também resultam em erros antes que gerem resultados inválidos para a aplicação.

2 FUN

FUN é uma linguagem de programação baseada em cálculo lambda (λ) que junta a tipagem lambda, com capacidade de modelar polimorfismo e OOP(Object Oriented Language). Esta linguagem encapsula conceitos já citados anteriormente.

Características da linguagem:

- Possui tipagem e também pode funcionar sem fazer tipagem explícita.
- Tipos parametrizados
- Abstração de tipos
- Herança
- Cinco tipos básicos diferentes