Resumo das duas primeiras seções do artigo "On Understanding Types, Data Abstraction, and Polymorphism"

Lucas Santiago

Maio de 2020

Resumo

A ideia desse artigo é resumir as duas primeiras seções do artigo *On Understanding Types, Data Abstraction, and Polymorphism* de Luca Cardelli e Peter Wegner. Com isso em mente, citarei tanto como funcionam os dados dentro de computadores, o porquê de tipos fortes serem positivos para os códigos. Além da história e evolução dos tipos de polimorfismo, concluindo em como funciona e quais os grandes problemas do Cálculo Lambda.

1 Representação de dados

A memória por si é apenas um conjunto de valores em binário, sem distinção entre caracteres, valores numéricos, ponteiros ou programas inteiros. A interpretação dos dados é feita de forma externa, por alguma função, por exemplo.

O nome dos tipos são criados ao conseguir unir um conjunto de valores por uma propriedade em comum. Por exemplo, inteiros são números representados sem parte decimal e números flutuantes são valores inteiros mais uma parte decimal.

A ideia de ter-se criado línguas de programação de tipagem forte é proteger a variável de ser interpretada de forma errônea, o uso, por exemplo, de um inteiro com um ponteiro poderia ser desastroso ao se acessar arbitrariamente os dados de outro programa em execução. Línguas estaticamente tipadas foram inventadas como uma proteção para o programador, mesmo línguas como python sem tipagem explícita (tipos só são específicos nela na hora dos *castings* de tipo) trabalham fortemente com a tipagem inferida mesmo sem o conhecimento do programador para proteger a aplicação.

2 Tipos de polimorfismo

Línguas com tipagem convencional, não suportam que uma mesma variável tenha mais de um tipo e são chamadas de monomórficas. Com a ideia de expandir o leque de possibilidades, foram criadas variáveis polimórficas. *Polimorfismo paramétrico* é quando alguma função consegue trabalhar em um conjunto específico de tipos ou estrutura de dados parecidos. *Polimorfismo Ad-Hoc* é mais abrangente conseguindo executar um conjunto de operações (ou pelo menos, tentando executar corretamente) em um conjunto de tipos e estruturas, mesmo que sem semelhanças entre si, reagindo de formas diferentes para cada tipo.

Polimorfismo Ad-Hoc é dividido em dois grandes grupos *overloading* e *coercion*. O primeiro é apenas uma simplificação para os programadores, permitindo que eles escrevam funções com mesmo nome, mas diferindo pelos argumentos ou retorno da função. Essas funções são renomeadas durante a hora da compilação. O segundo é uma conversão de tipos implícita feita para conseguir usar uma variável de um tipo em uma função que opera usando outros tipos.

Outras formas de polimorfismo são identificadas como o uso de coerção de tipos para operações simples, como somar um inteiro com um número real e resultar em um número real, sem a conversão explícita de tipos de inteiro para real antes da soma. Essa forma de polimorfismo não é considerada verdadeira, pois pode gerar ambiguidade, não é possível descobrir se um número é inteiro ou flutuante depois dessa soma, se não tiver o contexto anterior. A função

de somar não consegue dizer com antecedência se o resultado será inteiro ou flutuante antes da execução do programa. Para ser considerado polimorfismo verdadeiro é necessário que apenas 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.

3 História sobre polimorfismo

No início da computação, todos os valores tinham apenas um tipo aritmético. Foi apenas a partir do *FORTRAN* que começou a ideia de separar entre inteiros e pontos-flutuantes, uma vez que os hardwares começaram a se especializar e números inteiros eram executados mais rapidamente. *ALGOL 60* foi o primeiro a adicionar diferenças explícitas entre tipos de variáveis, podendo gerar erros durante a compilação por conta de tipagens incorretas, além de criar a ideia de escopo de variáveis.

Pascal usa tipagem explícita e forte, mas não define equivalência entre tipos diferentes. Com isso vários problemas como ambiguidades e problemas de segurança podem aparecer nos códigos. *ALGOL 68* tem uma noção de tipagem ainda mais forte que *Pascal* já contendo equivalência de tipos e estruturas.

Simula foi a primeira língua 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.

4 Cálculo Lambda (λ)

Cálculos lambda foram criados para uma mesma variável ou função suportar vários tipos ou estruturas. Dessa forma, uma função pode ser usada como uma

variável, permitindo que uma função seja executada em si mesma. 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, uma vez 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 tipar essas operações é necessário que algum compilador tenha 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 língua *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.