Avaliação 2 - 26 de março de 2021

Conceitos e Linguagens de Programação

Vinícius do Amaral Brunheroto

1)

a)

Na função aleph será printado os valores “10 39 25” e na função main será printado “10 15”

Como os parâmetros são passados por valor, x recebe 10, y recebe 10 e z recebe 25 na função aleph, porém dentro dela, y muda duas vezes de valor: inicialmente y é 15(10(y)+5) e depois 39(10(x)+25(z)+4). Por isso, de acordo com os últimos valores tende a printar 10(x), 39(y) e 25(z).

Já no main, será printado os valores que foram definidos junto às variáveis, como foram passados por valor, as mudanças dentro da função aleph não acarretam mudanças nos valores definidos no main para as variáveis i,j,k.

b)

Dentro de aleph, x e y recebem o endereço de j declarado no main e z recebe a cópia do valor de k declarado no main.

O conteúdo de y equivale a 44(15(x, pois ele aponta para o mesmo lugar que y)+25(z)+4).

No aleph, irá printar: “44 44 25”.

Então ao terminar a subrotina aleph, a variável j estará armazendo o valor 44, como x e y apontam para j, o conteúdo deles também considera essa mudança.

Ao voltar para o main, será printado: “44 15”.

Ou seja, as mudanças realizadas na subrotina aleph , graças à passagem por referência (endereço de j), interferiram nos valores de j do main.

Já a variável z, passada por valor, não teve seus valores alterados e mesmo que ocorresse mudanças dela no aleph, essas mudanças não se refletiriam no main.

c)

Na passagem por nome, dentro do Aleph , o j muda de valor duas vezes primeiro vai para 15(j=j+5--> j=10+5-->j=15) e depois para 49(j=j+(j+k)+4-->j=15+(15+15)+4-->j=49, então será printado “49 49 64”. O 64 é porque z= j+k-->z=49+15-->z=64

No main, será printado o valor de j e o de k: “49 15”.

Como é passagem por nome, há uma constante substituição de elementos pela ideia do nome.

2)

a)

A verificação gradual permite uma espécie de combinação(mistura) entre os aspectos da verificação estática e os aspectos da verificação dinâmica.

Então, pode ajudar uma linguagem que tem apenas verificação dinâmica a passar por uma etapa de verificação estática, a análise de tipos durante a compilação, auxiliando na descoberta de erros escondidos,melhorando o desempenho do programa.

Como exemplo, no caso do TypeScript, que é uma ferramenta que adiciona tipagem estática ao JavaScript que por padrão tem tipagem dinâmica.

A principal vantagem é o debug. O número de falhas no código reduz drasticamente, diminuindo o número de testes e debugs.

Outras vantagens são descobrir erros durante a etapa de desenvolvimento e incrementar a inteligência da IDE que está sendo usada. Sem ela, a IDE não poderia descobrir o formato de parâmetros e seria necessário entender métodos e valores que poderiam ser requisitados.

Além de ajudar na etapa de desenvolvimento, permite a utilização de funcionalidades da linguagem que não estão disponíveis na forma nativa.

No TypeScript, quando não é definido um tipo, o compilador usa o type inference

b)

Na verificação estática há métodos formais leves de validação, mais apoio a ferramentas e toda a lógica construída na compilação é mantida na execução, o que permite uma análise maior de outros programadores, de usuários.

O que se percebe na verificação gradual, é que métodos que são implementados em uma etapa são totalmente ignorados em outras etapas: por exemplo, no TypeScript, todo o código volumoso descrito para as tipagens não irá para a produção, é totalmente ignorado depois de compilado, então outros programadores e usuários não terão acesso a esse código de tipagens.

Além disso, a verificação gradual é menos conhecida que a verificação estática,por ser um conceito novo que vem sendo implementado, então pode-se dizer em uma mão de obra escassa para aplicação da verificação gradual.

3)

a) O poliformismo de subtipos ocorre quando a subclasse precisa modificar métodos que foram herdados da superclasse, então ela o sobreescreve usando override.

Quando há poliformismo de subtipos, acaba havendo mais de uma forma de vincular um nome a uma implementação.

Então deve existir maneiras de determinar qual implementação está vinculada a um nome.

Isso será feito por meio de vinculação dinâmica(dynamic binding) no momento da execução.

Para que se possa fazer a vinculação dinâmica, é necessário usar o dynamic dispatch, que é uma espécie de estrutura implementada "debaixo dos panos" que se baseia no switch de tipos, então dependendo do tipo, ele irá para uma implementação diferente.

As três implementações existem na memória, então o dynamic dispatch permite que haja uma espécie de "jump" para a implementação correspondente ao tipo.

b)

Os modificadores de visibilidade servem para regular o controle de acesso, o que pode ser exportado, o que pode ser visto e por quem pode ser visto e modificado.

Usando como referência a linguagem Kotlin, pode-se citar dois modificadores de visibilidade, como por exemplo PUBLIC, que é usado como default em Kotlin e indica que a a propriedade é visível em todo o lugar(ambiente) do código, por outro lado, PRIVATE indica que a propriedade só é visivel dentro do arquivo file indicado.

4)

As principais características que se manifestam para o uso Tipo Abstrato de Dados(TAD): são abstração, que se dá por meio de processos cognitivos, colocando em foco alguns aspectos em detrimento de outros, e a separação (independência) do código em relação aos detalhes.

Porém, há algo mais importante que serve para aplicar essas características: há um esquema que deve ser seguido, que está associado com as duas condições que o compilador tem que garantir numa linguagem de programação: primeiro, o cliente só pode usar e ter acesso a interface(abstração) para cumprir o papel de esconder, encapsular, ocultar os detalhes de implementações do cliente, segundo, as implementações precisam obedecer a interface, ou seja, seguir regras e definições que foram dadas na interface, para que possa seguir a mesma lógica.

Com isso, obtém-se uma relação entre cliente, interface e suas implementações e garante ao programador a possibilidade de usar Tipos Abstratos de Dados em seus sistemas.

5)

a) No paradigma imperativo, o programador diz COMO fazer, emite ordens, comandos, então o código representa uma lista de comandos.

É inspirado nas máquinas de Turing.

Exemplos de LP'S que seguem esse paradigma imperativo: C,C++.Java,Python..

No paradigma declarativo, o programador diz O QUE fazer, então emite definições, leis, enunciados.

Dentro do paradigma declarativo, há dois caminhos: funcional, que tem programas como coleções de funções matemáticas(inspirado no cálculo lambda λ),exemplos de lp's: Lisp,ML,Ocaml,Scala,Haskell, e o lógico que tem programas como coleções de relações matemáticas,exemplos de lp's: Prolog,Mercury..

b)

Os efeitos colaterais são algo de extrema importância para os programadores, pois o interesse da computação está nesses efeitos.

Até mesmo a programação funcional, não é "sem efeitos", mas sim eles são explicitamente representados e tratados, sendo confinados em parte do código.

A promessa da programação funcional é poder modelar esses efeitos e poder fazer composições.

Por isso as linguagens de programação já acabam tendo mecanismos de tal forma a controlar e gerenciar esses efeitos, quando necessário,para evitar alguns desses efeitos como possibilidades de erro, contornar exceções. Como exemplo, em kotlin, que usa "?." para compor e ao mesmo tempo considerar a existência de efeitos colaterais em funções.