Sistemas de tipos- relações entre tipos

Dados dois tipos T e U consideram-se as possíveis relações entre eles:

T está contido em U (T < U) – O conjunto de valores possíveis em T é um subconjunto dos valores de U

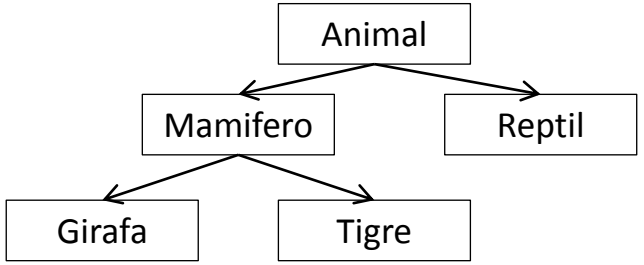
T contém U (T > U)) – O conjunto de valores possíveis em U é um subconjunto dos valores de T

T igual U (T = U) – T e U possuem o mesmo conjunto de valores

T não está relacionado com U (T <> U) – Os valores de U e T não estão relacionados

Por exemplo, dada a hierarquia dos tipos `Animal`, `Mamifero`, `Girafa`, `Tigre`, `Reptil`, temos:

```
Mamifero < Animal
Mamifero > Girafa
Girafa <> Tigre
Girafa <> Reptil
```



Os tipos T e U **não** têm de estar hieraquicamente relacionados. Por exemplo `double > float` e `int > char`.

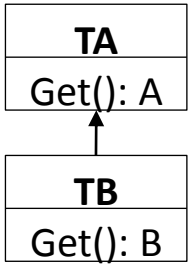
Nos sistemas de tipos, dados dois tipos T e U define-se o princípio, conhecido como **princípio da substituição**:

se T > U, então onde se espera uma instância de T pode ser usada uma instância de U. Exemplos:

```
Mamifero m = new Girafa();
float f = …;
double d = f;
```

Variância- covariância e contravariância

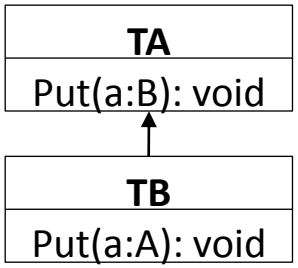
Sejam os pares de tipos (T,U) e (T',U'). Uma dada "transformação" diz-se **covariante** quando, aplicada a T e U, com determinada relação, resulta em dois tipos T', U', que mantêm essa mesma relação. Exemplificando:



- Sejam as classes TA e TB (onde TA > TB). A retorno do método Get nas duas classes é covariante se A > B.
  Seja `TB tb … ; TA ta = tb;  A a = ta.Get();`
  Uma vez que Get de TB retorna um B, este também pode ser afectado a uma variável do tipo A.

Sejam os pares de tipos (T,U) e (T',U'). Uma dada "transformação" diz-se **contravariante** quando, aplicada a T e U, com determinada relação de inclusão, resulta em dois tipos T', U', que invertem essa mesma relação de inclusão. Exemplificando:

- Sejam as classes TA e TB (onde TA > TB). A definição do método Put nas duas classes é contravariante se B < A.
  Seja `TB tb … ; TA ta = tb;  B b = …; ta.Put(b);`
  Uma vez que Put de TB recebe um A, pode também receber um B.

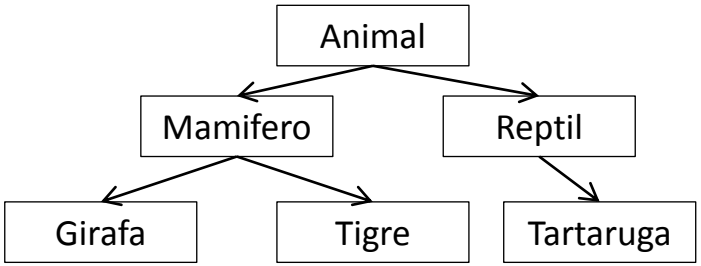


Nota: este tipo de variância, em parâmetros e retorno de métodos virtuais não existe no Common Type System.

Variância no CTS e em C#- covariância em arrays

- Desde o primeira especificação do CLI (e do C#) os arrays de tipos referência são covariantes, ou seja:

  Dados dois tipos referência T, U onde `T > U`, então `T[] > U[]`. Por exemplo, é possível escrever `Animal[] animais = new Girafa[10];`



A covariância de arrays tem um problema fundamental. Uma vez que um elemento de `Animal[]` pode ser afectado com qualquer *animal* então podemos fazer `animais[0] = new Tartaruga().` O compilador aceita a afectação como correcta.

A única forma de evitar a afectação é efectuar sempre o teste em *runtime* e lançar uma excepção. É exactamente isso que faz o CLR. Ou seja, a covariância em arrays gerou uma penalização na afectação de elementos de *arrays* de tipos referência, uma vez que não se pode garantir à partida que o objecto referido seja de tipo compatível.

Variância no CTS e em C#- variância na associação de métodos a *delegates*

Seja a seguinte definição de um tipo delegate em C#:

```
delegate Animal  Collect()
```

Suponha o método:

```
static Giraffe MakeGiraffe()
```

É possível fazer:

```
Collect  func = MakeGiraffe;
```

Este tipo de **covariância** é similar à apresentada no tipos de retorno de métodos, no contexto da redefinição de métodos virtuais. O chamador de func espera um qualquer *animal* pelo que pode receber uma *girafa*.

Seja agora o delegate: `delegate void Release(Mamifero m)`

Suponha as seguintes definições:

```
static void joinHerd(Giraffe g) {…}
static void putInFreedom(Animal a) {….}
Releaser action1 = joinHerd;        // ilegal, na chamada pode passar um mamífero diferente!
Releaser action2 = putInFreedom;    // legal, qualquer animal serve!
```

Como no caso acima, este tipo de **contravariância** é similar à apresentada nos parâmetros de entrada de métodos virtuais. Na chamada `action1` e `action2` pode ser passado qualquer *mamifero*, pelo que um método que espera *animal* consegue lidar com Mamifero.

Variância no C# 4.0- variância em interfaces e *delegates* genéricos

Até ao C# 3.0, embora fosse possível a variância no binding de métodos a delegates, como explicada no slide anterior, não existia variância entre tipos delegates. Por exemplo, seja o delegate genérico

```
delegate U Func<T,U>(T t).
```

No código seguinte, atendendo a covariância no tipo de retorno e à contravariância nos parâmetros de entrada, podemos dizer que `Func<Mamifero, Mamifero> > Func<Animal, Girafa>`. Embora o CTS desde sempre tenha suportado este tipo de variância, o compilador do C#, até a versão 3.0 da linguagem, queixa-se da afectação `f2=f1`, pois os tipos *delegate* eram considerados invariantes.

```
Func<Animal, Giraffe> f1 = …;  // delegate que recebe um animal e retorna girafa
Func<Mammal, Mammal> f2 = f1;
```

Na versão 4.0 da linguagem, *delegates* e interfaces genéricas passaram a ter variância nos tipos dos parâmetros e do retorno, usando-se as keywords **`in`** e **`out`** para especificar, respectivamente, **contravariância** e **covariância**.

Definindo o delegate anterior como `delegate U Func<in T, out U>(T t)` o código mostrado acima passa a compilar sem erros na versão 4.0 da linguagem C#.

Exemplo de variância com interfaces. Seja o método `FeedAnimals`. Em C# 4.0, podemos escrever:

```
void FeedAnimals(IEnumerable<Animal> animals){
    foreach(Animal animal in animals)
        if (animal.Hungry)
            Feed(animal);
}
```

```
List<Giraffes> girafas = …;
IEnumerable<Giraffe> adultas= girafas.Where( g => g.Age > 5);
FeedAnimals(adultas);
```

Pois a interface `IEnumerable<T>` passou a ser covariante no tipo `T` (retorno de `GetEnumerator`).

Referências

- Os slides foram baseados numa série de artigos de Eric Lippert sobre variância disponível em:

http://startbigthinksmall.wordpress.com/2010/05/26/resources-on-covariance-and-contravariance/