Fun with Generics

Benjamin Hodgson

17th September 2015

Example 1: FileStore

```
interface IFileStore
{
    Stream Read(IPathKey key);
    void Write(IPathKey key, Stream stream);
}
```

```
class FileSystemFileStore : IFileStore
{
    IFileSystemPathGenerator _generator;
    IFileSystemReader _reader;
    IFileSystemWriter _writer;

    public Stream Read(IPathKey key)
    {
        string absolutePath = _generator.Generate(key);
        return _reader.Read(absolutePath);
    }
    public void Write(IPathKey key, Stream stream)
    {
        string absolutePath = _generator.Generate(key);
        _writer.Write(absolutePath, stream);
    }
}
```

```
class AwsFileStore : IFileStore
{
    IAwsKeyGenerator _keyGenerator;
    IAwsReader _reader;
    IAwsWriter _writer;

    public Stream Read(IPathKey key)
    {
        S3Key ey = _keyGenerator.Generate(key);
        return _reader.Read(key);
    }
    public void Write(IPathKey key, Stream stream)
    {
        S3Key key = _keyGenerator.Generate(key);
        _writer.Write(key, stream);
    }
}
```

```
class RackspaceCloudFileStore : IFileStore
{
    IRackspaceContainerNameGenerator _cGenerator;
    IRackspaceObjectPathGenerator _pathGenerator;
    IRackspaceReader _reader;
    IRackspaceWriter _writer;
    public Stream Read(IPathKey key)
    {
        string containerName = _cGenerator.Generate(key);
        string objectPath = _pathGenerator.Generate(key);
        return _reader.Read(containerName, objectPath);
    }
    public void Write(IPathKey key, Stream stream)
    {
        string containerName = _cGenerator.Generate(key);
        string objectPath = _pathGenerator.Generate(key);
        _writer.Write(containerName, objectPath, stream);
    }
}
```

```
interface IPathGenerator<out TPath>
{
    TPath Generate(IPathKey key);
}
interface IFileReader<in TPath>
{
    Stream Read(TPath path);
}
interface IFileWriter<in TPath>
{
    void Write(TPath path, Stream stream);
}
```

And:

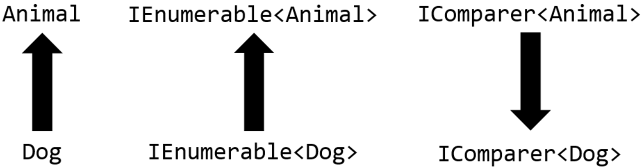
```
class RackspacePath
{
    public string ContainerName { get; set; }
    public string ObjectPath { get; set; }
}
```

```
class FileStore<TPath> : IFileStore
{
    IPathGenerator<TPath> _pathGenerator;
    IFileReader<TPath> _reader;
    IFileWriter<TPath> _writer;

    public Stream Read(IPathKey key)
    {
        TPath path = _pathGenerator.Generate(key);
        return _reader.Read(path);
    }
    public void Write(IPathKey key, Stream stream)
    {
        TPath path = _pathGenerator.Generate(key);
        _writer.Write(path, stream);
    }
}
```

Variance

When is a generic type a subtype of another generic type?



- IList<T> is *invariant*
  - An IList is never a subtype of another IList
  - T appears in both *input* and *output* positions
- IEnumerable<out T> is *covariant*
  - IEnumerable's subtyping relation goes in the *same direction* as the type parameter
  - IEnumerable *produces* Ts
- IComparer<in T> is *contravariant*
  - IComparer's subtyping relation goes in the *opposite direction* to the type parameter
  - IComparer *consumes* Ts

Variance

```
float AverageAge(IEnumerable<Animal> animals)
{
    return animals.Select(a => a.Age).Sum()
           / animals.Count();
}
AverageAge(new List<Dog> { rex, fido, richard });
```

Variance

```
float AverageAge(IEnumerable<Animal> animals)
{
    return animals.Select(a => a.Age).Sum()
           / animals.Count();
}
AverageAge(new List<Dog> { rex, fido, richard });
```

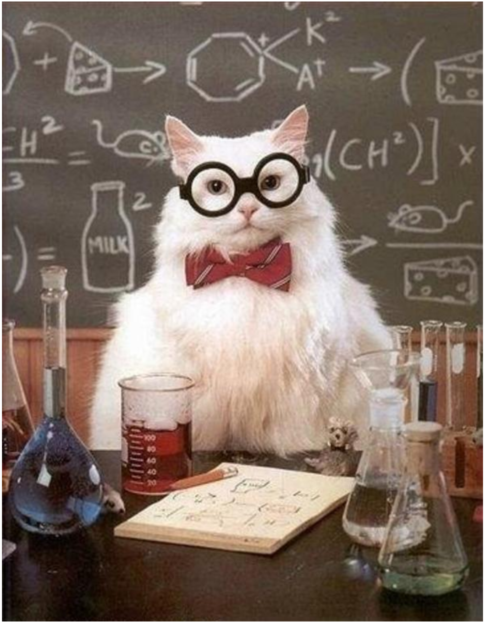
```
bool DogIsBetter(IComparer<Dog> comparer)
{
    return comparer.Compare(this.dog1, this.dog2) > 0;
}
DogIsBetter(new AnimalComparer());
```

Variance

```
float AverageAge(IEnumerable<Animal> animals)
{
    return animals.Select(a => a.Age).Sum()
           / animals.Count();
}
AverageAge(new List<Dog> { rex, fido, richard });

bool DogIsBetter(IComparer<Dog> comparer)
{
    return comparer.Compare(this.dog1, this.dog2) > 0;
}
DogIsBetter(new AnimalComparer());

void AddGiraffe(IList<Animal> animals)
{
    animals.Add(new Giraffe());
}
// AddGiraffe(new List<Dog>());
```


MILK

The Spectrum...

- ... of Safety
- ... of Usability
- ... of Power
- ... of Flexibility

Example 2: Permissions Engine

Specifications: small classes testing something about an object, and a way to combine them.

```
interface ISpecification<in T>
{
    bool IsSatisfiedBy(T candidate);
}
```

Example 2: Permissions Engine

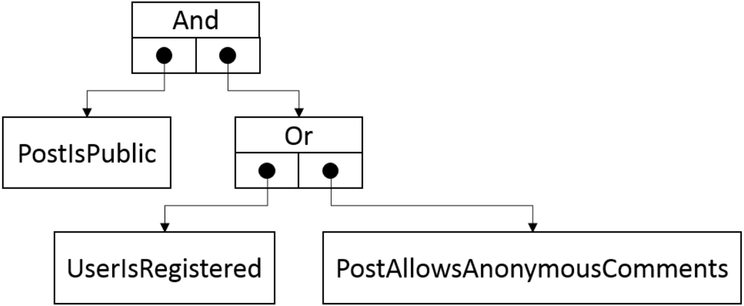
Specifications: small classes testing something about an object, and a way to combine them.

```
interface ISpecification<in T>
{
    bool IsSatisfiedBy(T candidate);
}
```

```
var context = new BlogContext
{
    CurrentUser = user,
    BlogPost = post
}
if (!userCanCommentOnBlogPost.IsSatisfiedBy(context))
{
    throw new PermissionException(
        "You can't comment on that post"
    );
}
```

Example 2: Permissions Engine

Represent combinations of specifications as a syntax tree.




```
var userCanCommentOnBlogPost =
  new AndSpecification<BlogContext>(
    new PostIsPublic(),
    new OrSpecification<BlogContext>(
      new UserIsRegistered(),
      new PostAllowsAnonymousComments()
    )
);
```

Example 2: Permissions Engine

```
interface ISpecificationVisitor<T, out TReturn>
{
    TReturn Visit(LeafSpecification<T> spec);
    TReturn Visit(AndSpecification<T> spec);
    TReturn Visit(OrSpecification<T> spec);
    TReturn Visit(NotSpecification<T> spec);
}
interface ISpecification<T>
{
    TReturn Accept<TReturn>(
        ISpecificationVisitor<T, TReturn> visitor);
}
```

Problem

Code duplication because `void` is not a real type

```
interface ISpecificationVisitor<T>
{
    void Visit(LeafSpecification<T> spec);
    void Visit(AndSpecification<T> spec);
    void Visit(OrSpecification<T> spec);
    void Visit(NotSpecification<T> spec);
}
```

Alternatively, a type that means the same as `void`:

```
sealed class Unit
{
    private static readonly Unit _default = new Unit();
    private Unit() { }
    public Unit Default { get { return _default; } }
}
```

Problem

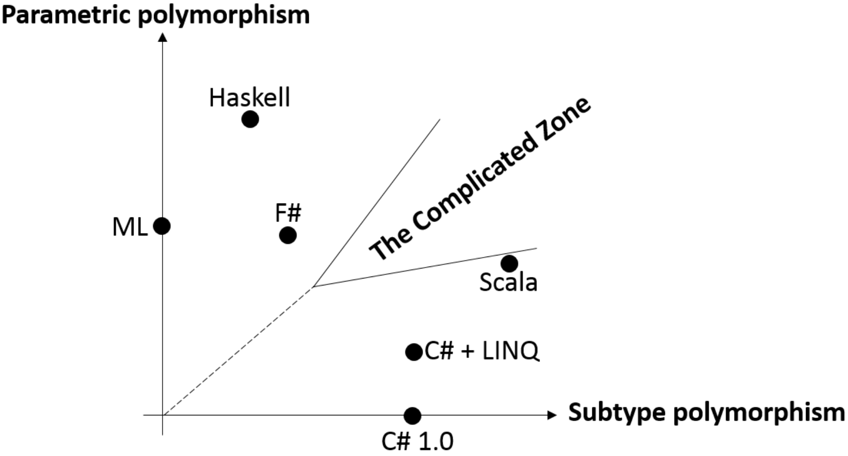
Why isn't `ISpecification<T>` contravariant? A specification which tests fruit should work when you need to test an apple.

```
class FruitIsRipe : ISpecification<Fruit> { /* ... */ }
void TestApple(ISpecification<Apple> spec);
```

- `ISpecificationVisitor<T, TReturn>` would have to be *co*variant in `T`
- `LeafSpecification<T>`, `AndSpecification<T>` etc would have to be *contra*variant
- Classes can never be variant in C♯
- You have to make an interface for every class!

Two kinds of polymorphism

Parametric polymorphism	Subtype polymorphism
Lots of generality	Only works with subtypes
Requires design-time foresight	Ad-hoc extensibility



Example 3: Test Data Builders

- A simple tool to make it easy to set up test data – everything gets a default, which can be overridden if the test needs it.
- Imagine testing a the `SalaryPayment` class of a payroll system
  - When *building*, the recipient of a payment gets a default value
  - When *saving*, you have to set the recipient – it must be a *real* `Employee` to satisfy the foreign key

```
new PaymentTestDataBuilder()
    .WithRecipient(employee)
    .Save();  // fine
new PaymentTestDataBuilder()
    .Build();  // fine
new PaymentTestDataBuilder()
    .Save();  // runtime exception!
```

Attempt 1: Subclass

```
class PaymentTestDataSaver : PaymentTestDataBuilder
{
    public PaymentTestDataSaver(Employee recipient)
    { /* ... */ }
    public SalaryPayment Save()
    { /* ... */ }
}

new PaymentTestDataSaver(employee)
    .Save();  // :)
new PaymentTestDataSaver(employee)
    .WithDateIssued(new DateTime(2015, 9, 17))
    .Save();  // compile error :(
```

The declared return type of `WithPaymentDate` is `SalaryPaymentTestDataBuilder`.

Attempt 2: Parameterise the hierarchy

Idea: Parametrise the return type and specialise in the subclasses. Use *F-bounds* to constrain the return type to be "the type of `this`".

```
abstract class PaymentTestDataBuilder<TSelf>
    where TSelf : PaymentTestDataBuilder<TSelf>
{
    public TSelf WithRecipient(Employee recipient)
    {
        // ...
        return This();
    }
    public TSelf WithDateIssued(DateTime date)
    {
        // ...
        return This();
    }
    protected abstract TSelf This();
}
```

Attempt 2: Generic base class

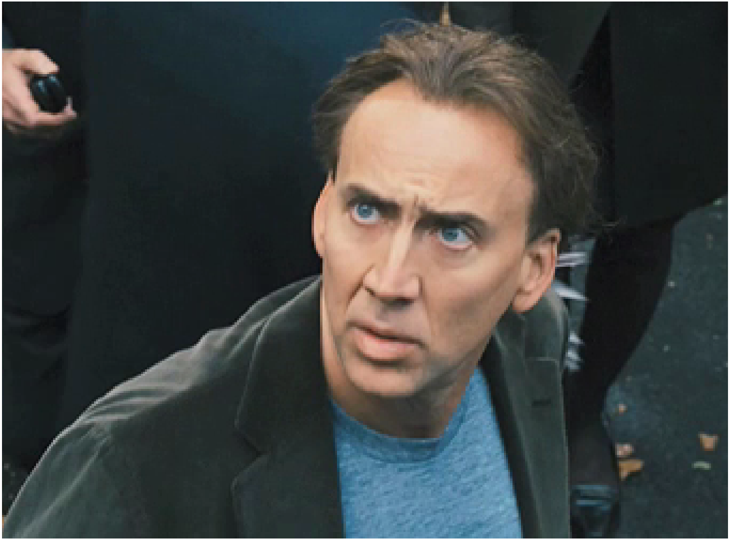
```
class PaymentTestDataBuilder :
    PaymentTestDataBuilder<PaymentTestDataBuilder>
{
    protected override PaymentTestDataBuilder This()
    { return this; }
}
class PaymentTestDataSaver :
    PaymentTestDataBuilder<PaymentTestDataSaver>
{
    public PaymentTestDataSaver(Employee recipient)
    { /* ... */ }
    protected override PaymentTestDataSaver This()
    { return this; }
    public SalaryPayment Save() { /* ... */ }
}
```

Problems

F-bounds aren't quite strong enough.

```
class PathologicalBuilder
                                        // Not the type of this!
    : PaymentTestDataBuilder<PaymentTestDataSaver>
{
    // ...
}
```

Once you have one valid subclass, you can add invalid subclasses to your heart's content. F-bounds are not *quite* strong enough to express self-types.



Attempt 3: Type-level data

Idea: use *phantom types* to represent validity

```
sealed class True { }
sealed class False { }

class PaymentTestDataBuilder<HasRecipient>
{
  public PaymentTestDataBuilder<True>
        WithRecipient(Employee recipient)
  {
    return new PaymentTestDataBuilder<True>
        (recipient);
  }
}
```

Attempt 3: Type-level data

```
class PaymentTestDataBuilder
    : PaymentTestDataBuilder<False>
{ }

static class TestDataBuilderExtensions
{
    public SalaryPayment Save(
        this PaymentTestDataBuilder<True> builder)
    {
        // ...
    }
}
```


COPYRIGHT MARTA TESORO 2012

Example 4: Vectors

The classic toy example of dependent types: a sequence with static knowledge of how long it is.

Idea: Write a phantom type representing lengths. Increment this phantom type every time you add an element to a vector. Later, check whether the length type parameter is greater than 1.

Example 4: Vectors

The classic toy example of dependent types: a sequence with static knowledge of how long it is.
Idea: Write a phantom type representing lengths. Increment this phantom type every time you add an element to a vector. Later, check whether the length type parameter is greater than 1.

How to represent numbers in the type system? Mathematicians define natural numbers inductively:

1. 0 is a natural number.
2. For every natural number n, the *successor* of n, $S(n)$, is a natural number.

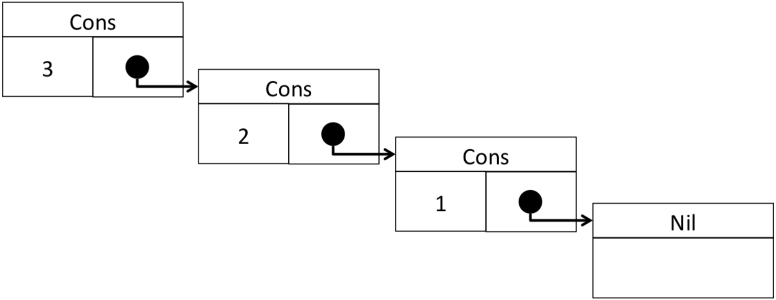
```
sealed class Z { }  // Z for Zero
sealed class S<N> { }  // S for Successor
```

0 $\leftrightarrow$ `Z`
1 $\leftrightarrow$ `S<Z>`
2 $\leftrightarrow$ `S<S<Z>>`
3 $\leftrightarrow$ `S<S<S<Z>>>`

Linked list recap

- Two cases: `Nil` and `Cons`.
  - `Nil` contains no elements.
  - `Cons` contains one element and a pointer to the rest of the list.
- Represent each case as a class.

`[3,2,1]` $\leftrightarrow$ `new Cons(3, new Cons(2, new Cons(1, Nil)))`



Example 4: Vectors

Implement Vec like a linked list:

```
abstract class Vec<N, T> { }
sealed class Nil<T> : Vec<Z, T> { }
sealed class Cons<N, T> : Vec<S<N>, T>
{
    public T Head { get; private set; }
    public Vec<N, T> Tail { get; private set; }
    public Cons(T elem, Vec<N, T> tail)
    {
        Head = elem;
        Tail = tail;
    }
}
```

Example 4: Vectors

```
static class VecExtensions
{
    public static T First<N, T>(this Vec<S<N>, T> v)
    {
        return ((Cons<N, T>)v).Head;  // :(
    }
    public static T Single<T>(this Vec<S<Z>, T> v)
    {
        return v.First();
    }
}
```

We know that the only way to get a `Vec` longer than 0 is if it's a `Cons`, but the language can't assume that because subclassing is *open*. You're forced to squeeze your ideas into an unsuitable subtyping relationship.

Problems

Annoying to manually specify all your types:

```
var v = new Cons<S<S<Z>>, int>(2,
    new Cons<S<Z>, int>(3,
        new Cons<Z, int>(7, new Nil<int>())
    )
);
```

Problems

Annoying to manually specify all your types:

```
var v = new Cons<S<S<Z>>, int>(2,
    new Cons<S<Z>, int>(3,
        new Cons<Z, int>(7, new Nil<int>())
    )
);
```

Luckily C♯ does (some) type inference of generic methods:

```
static Vec<S<N>, T> Cons<N, T>(T x, Vec<N, T> v)
{
    return new Cons<N, T>(x, v);
}
Cons(2, Cons(3, Cons(7, new Nil<int>())));
```

More problems

Problem: You can build a vector but you can't consume one — C♯ is too dumb to infer the type of the tail of a vector.

```
public static int Sum(Vec<Z, int> v)
{
    return 0;
}
public static int Sum<N>(Vec<S<N>, int> v)
{
    var c1 = (Cons<N, int>)v;
                    // compile failure :(
    return c1.Head + Sum(c1.Tail);
}
Sum(Cons(2, Cons(3, new Nil<int>())));
```

99 problems but a Vec ain't one. Actually that's not true

Type syonyms would be useful

```
class Three : S<S<S<Z>>> { }
class Four : S<Three> { }  // not the same as S<S<S<S<Z>>>>
```

No way to teach the type checker how to manipulate numbers - *you can't compute with types*.

```
Vec</* Plus<N, M>? */, T> Extend<N, M, T>(
    Vec<N, T> v1,
    Vec<M, T> v2)

Vec<N, T> Take<N, M, T>(N length, Vec<M, T> v2)
    // where LEq<N, M>?
```

Here's what you could've won

```
data Vect : Nat -> Type -> Type where
    Nil : Vect Z a
    (::) : a -> Vect n a -> Vect (S n) a

head : Vect (S n) a -> a
head (x :: xs) = x

sum : Vect n Int -> Int
sum Nil = 0
sum (x :: xs) = x + sum xs

(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil ys = ys
(++) (x :: xs) ys = x :: extend xs ys

take : n -> Vect (n + m) a -> Vect n a
take Z xs = Nil
take (S k) (x :: xs) = x :: take k xs
```

The End

- Types can be part of your arsenal in the battle against bugs
- Generics give you a lot of generality and type-safety at little expense
- C♯ has plenty of room for improvement. I would like to see at least some of:
  - Variant classes
  - Anonymous subclasses
  - Proper type inference
  - Closed types and pattern matching
  - Type syonyms
  - Higher-kinded types
  - Type functions
- I've barely scratched the surface of crazy type systems available in today's programming languages