Functional Programming From First Principles

Closing over the loop variable considered harmful

Eric Lippert 12 Nov 2009 6:50 AM 134

RATE THIS

(This is part one of a two-part series on the loop-variable-closure problem. Part two is here.)

UPDATE: We **are** taking the breaking change. In C# 5, the loop variable of a foreach will be logically inside the loop, and therefore **closures will close over a fresh copy of the variable each time**. The "for" loop will not be changed. We return you now to our original article.

I don't know why I haven't blogged about this one before; this is the single most common incorrect bug report we get. That is, someone thinks they have found a bug in the compiler, but in fact the compiler is correct and their code is wrong. That's a terrible situation for everyone; we very much wish to design a language which does not have "gotcha" features like this.

But I'm getting ahead of myself. What's the output of this fragment?

```
var values = new List<int>() { 100, 110, 120 };
var funcs = new List<Func<int>>();
foreach(var v in values)
  funcs.Add( ()=>v );
foreach(var f in funcs)
  Console.WriteLine(f());
```

Most people expect it to be 100 / 110 / 120. It is in fact 120 / 120 / 120. Why?

Because `()=>v` means "return **the current value of variable v**", not "return the value v was back when the delegate was created". **Closures close over variables, not over values.** And when the methods run, clearly the last value that was assigned to v was 120, so it still has that value.

This is very confusing. The correct way to write the code is:

Where is the loop variable declared?

```
foreach(var i in new[]{0,1,2,3,4})
{
   Console.WriteLine(i);
}
```

```
var i_ = default(int);
foreach(var i in new[]{0,1,2,3,4})
{
   i_ = i;
   Console.WriteLine(i_);
}
```

Outside?

```
foreach(var i in new[]{0,1,2,3,4})
{
   var _i = i ;
   Console.WriteLine(_i);
}
```

Inside?

Where is the loop variable declared?

```
0
1
2
3
4
```

```
foreach(var i in new[]{0,1,2,3,4})
{
   Console.WriteLine(i);
}
```

```
0
1
2
3
4
```

```
var i_ = default(int);
foreach(var i in new[]{0,1,2,3,4})
{
   i_ = i;
   Console.WriteLine(i_);
}
```

Who cares?

```
0
1
2
3
4
```

```
foreach(var i in new[]{0,1,2,3,4})
{
   var _i = i ;
   Console.WriteLine(_i);
}
```

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
   fis.Add(delegate{ Console.WriteLine(i);});
}
foreach(var fi in fis) fi();
```

Let’s capture it and see what happens

```
var fis = new List<Action>();
var i_ = default(int);
foreach(var i in  new[]{0,1,2,3,4})
{
    i_ = i;
    fjs.Add(delegate{ Console.WriteLine(i_);});
}
foreach(var fi in fis) fi();
```

Outside?

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
    var _i = i;
    fis.Add(delegate{ Console.WriteLine(_i);});
}
foreach(var fi in fis) fi();
```

Inside?

C#4

4
4
4
4
4

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
   fis.Add(delegate{ Console.WriteLine(i);});
}
foreach(var fi in fis) fi();
```

C#5

0
1
2
3
4

4
4
4
4
4

```
var fis = new List<Action>();
var i_ = default(int);
foreach(var i in  new[]{0,1,2,3,4})
{
    i_ = i;
    fjs.Add(delegate{ Console.WriteLine(i_);});
}
foreach(var fi in fis) fi();
```

0
1
2
3
4

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
    var _i = i;
    fis.Add(delegate{ Console.WriteLine(_i);});
}
foreach(var fi in fis) fi();
```

```
var fis = new List<Action>();
for(var i = 0; i < 5; i++)
{
   fis.Add(delegate{ Console.WriteLine(i);});
}
foreach(var fi in fis) fi();
```

```
5
5
5
5
5
```

```
var fis = new List<Action>();
var i_ = default(int);
for(i_ = 0; i_ < 5; i_++)
{
   fjs.Add(delegate{ Console.WriteLine(i_);});
}
foreach(var fi in fis) fi();
```

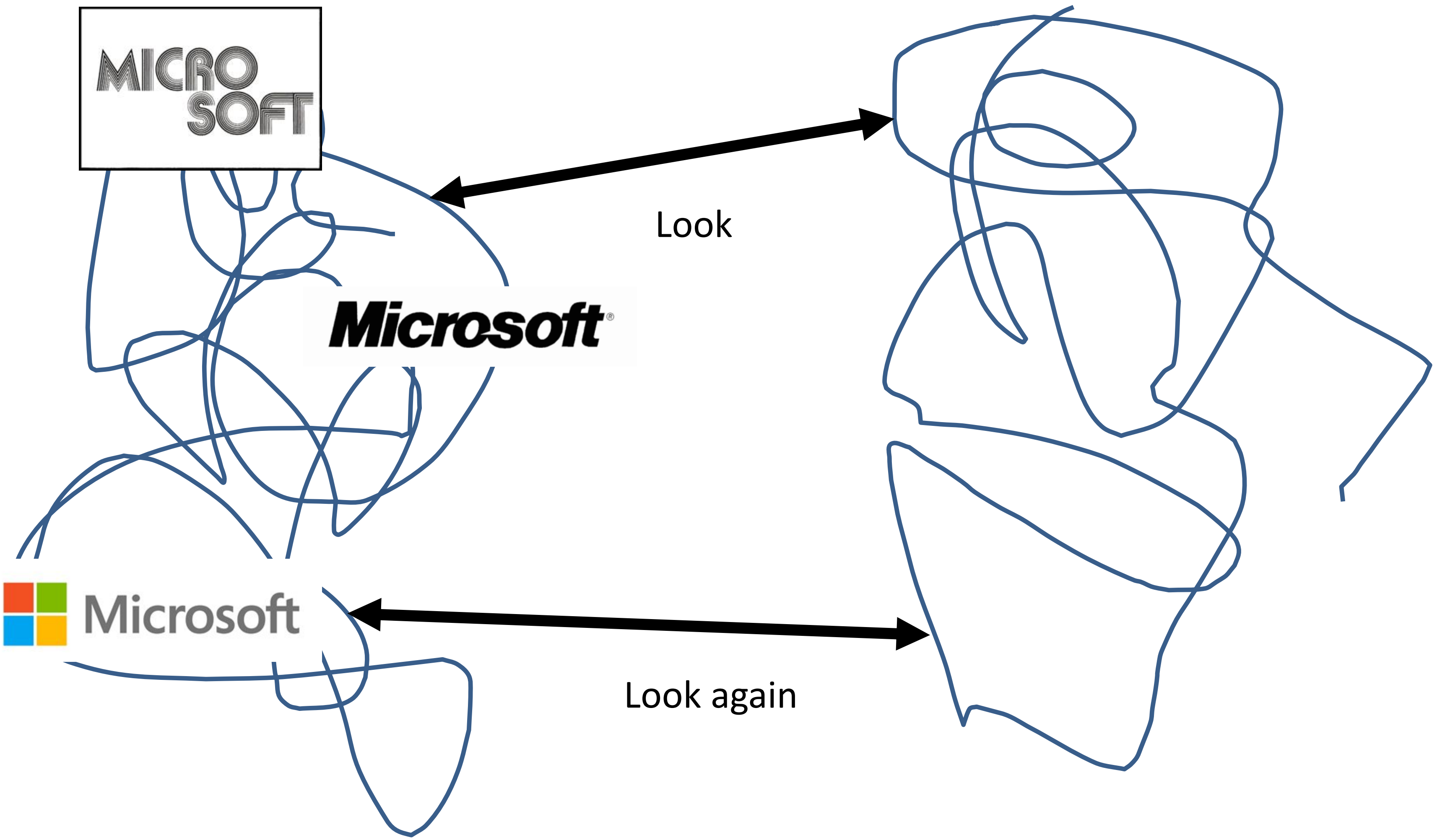
```
5
5
5
5
5
```

```
var fis = new List<Action>();
for(var i = 0; i < 5; i++)
{
   var _i = i;
   fls.Add(delegate{ Console.WriteLine(_i);});
}
foreach(var fi in fis) fi();
```

```
0
1
2
3
4
```

Who Gets The Blame?

The Real World is Imperative



Still don’t believe me?!

This monkey is now dead ….
May it rest in peace.

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
    fis.Add(delegate{ Console.WriteLine(i);});
}
foreach(var fi in fis) fi();
```

innocent

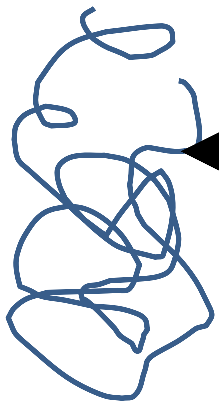
guilty

```
var fis = new List<Action>();
foreach(var i in  new[]{0,1,2,3,4})
{
    fis.Add(delegate{ Console.WriteLine(i);});
}
foreach(var fi in fis) fi();
```

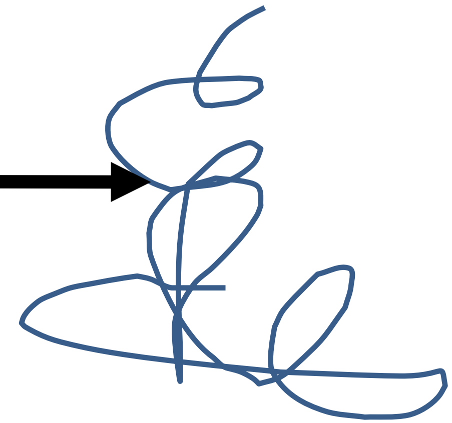
innocent

Acknowledge the presence of side-effects

producer



consumer



`0, 1, 2, 3, 5, …`

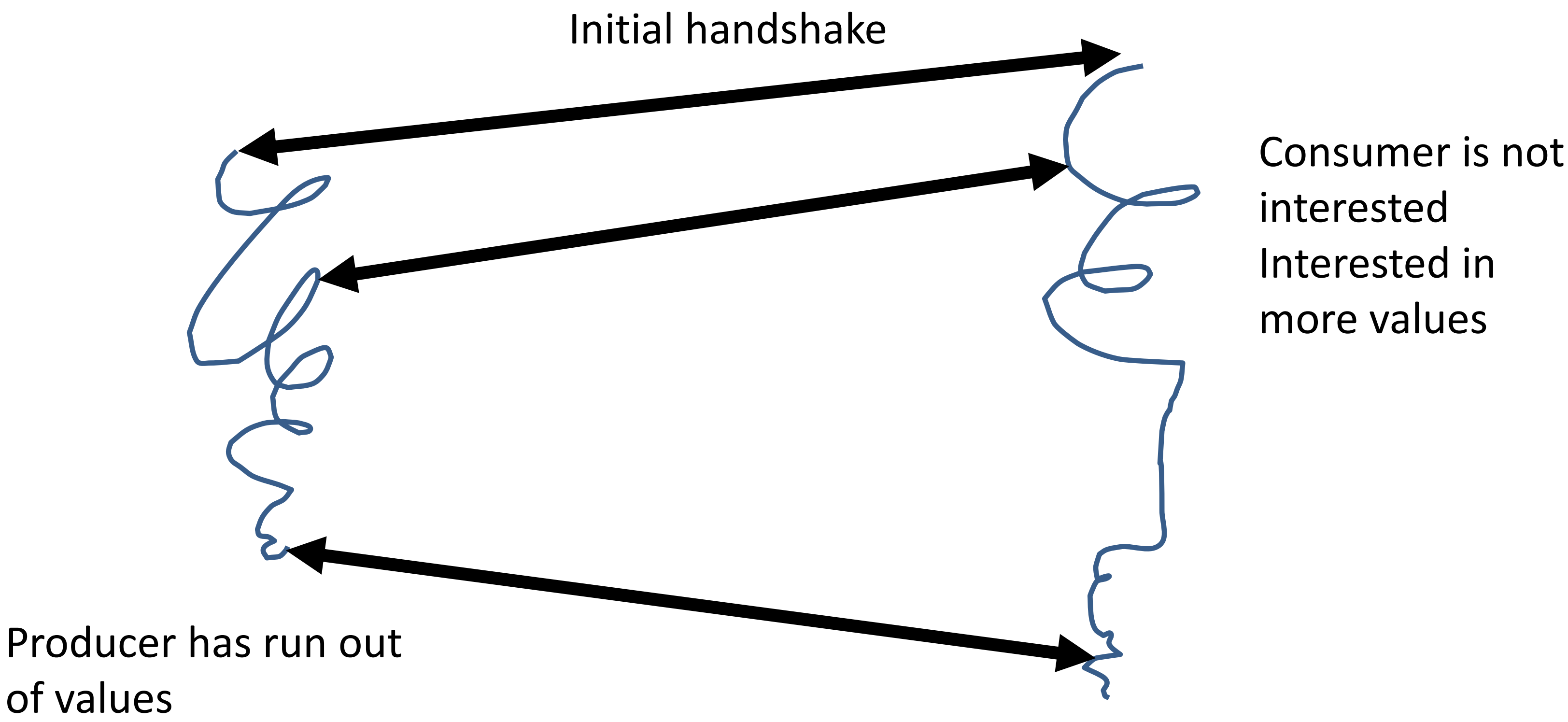
(0) read value mutable variable of type T

(1) If you want to see next value
call this `Func<T>` (i.e. `()`→T) to get it.

(2) Here is an `Action<T>` (i.e. T→`()`), notify me when the next value is available.

How to communicate a stream of values between producer and consumer?

Operational Details



Pull-based protocol (consumer asks for values)

```
interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

Initial handshake

```
interface IEnumerable<T> : IDisposable
{
    bool MoveNext();
    T Current { get; }
}
```

Call when want next value
()→T+()+Exception;

```
interface IDisposable
{
    void Dispose();
}
```

I won't bother you anymore,
you may forget about me.

Push-based protocol
(consumer gets notified of values)

```
interface IObservable<T>
{
   IDisposable Subscribe(IObserver<T> observer);
}
```

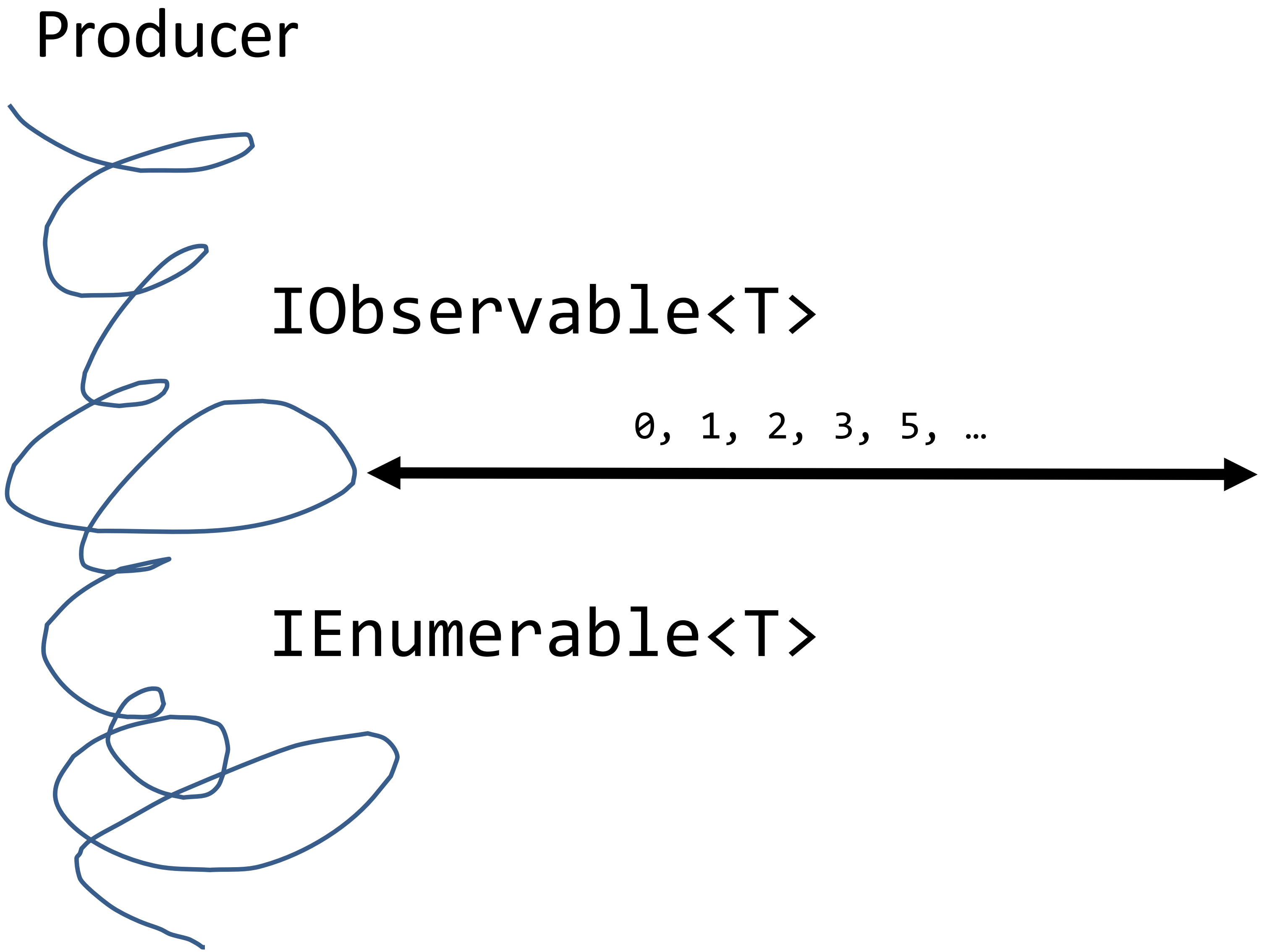
Initial handshake

```
interface IObserver<T>
{
   void OnNext(T value);
   void OnCompleted();
   void OnError(Exception error);
}
```

Notify when next value available
T+()+Exception→();

```
interface IDisposable
{
   void Dispose();
}
```

I won’t bother you anymore,
you may forget about me.


Producer
IObservable<T>
0, 1, 2, 3, 5, …
IEnumerable<T>

What if precisely one value?

```
class Lazy<T>
{
   T Value { get; }
}
```

Pull

```
class Task<T>
{
   Task<S> ContinueWith<S>
      (Func<Task<T>,S> continuation){ … }
   T Result{ get; }
}
```

Push

(Note concrete classes, not interfaces ☹)

五香粉

	One	Many
Pull	`T/Lazy<T>`	`IEnumerable<T>`
Push	`Task<T>`	`IObservable<T>`

The formulae are based on the Chinese philosophy of balancing the yin and yang in food. [Wikipedia]

Five-spice powder

From Wikipedia, the free encyclopedia

This article is about Chinese five-spice

Five-spice powder is a mixture of five spic

cookery.[1]

Contents [hide]

Five-spice powder

Chinese	五香粉
Hanyu Pinyin	wǔxiāngfěn
Transcriptions	[show]

Formulae [edit]

The formulae are based on the Chinese philosophy of balancing the yin and yang in food. There are many variants. The most common is *bajiao* (star anise), cloves, cinnamon, *huajiao* (Sichuan pepper) and ground fennel seeds.[2] Instead of true cinnamon, "Chinese cinnamon" (also known as *rougui*, the ground bark of the cassia tree, a close relative of true cinnamon which is often sold as cinnamon), may be used. The spices need not be used in equal quantities.[2]

Another variant is *tunghing* or "Chinese cinnamon" (powdered cassia buds), powdered star anise and anise seed, ginger root, and ground cloves.

In Chinese philosophy, the concept of yin-yang (simplified Chinese: 阴阳; traditional Chinese: 陰陽; pinyin: yīnyáng), which is often referred to in the West as "yin and yang", literally meaning "shadow and light", is used to describe how polar opposites or seemingly contrary forces are interconnected and interdependent in the natural world, and how they give rise to each other in turn in relation to each other. [Wikipedia]

Interface versus Implementation

Make all assumptions explicit

```
IEnumerable<T>
```

interface

Essence

```
T[]
List<T>
HashTable<K,T>
…
```

concrete class

Implementation details

In most OO languages the distinction is blurred

```
interface IA {}
abstract class A {}
static class B {}
sealed class C {}
class D
{ protected private virtual partial
   Foo Bar()
   { … this … }
}
```

```
foldr :: (a → b → b)
      → b
      → ([a] → b)
```

Concrete type
==
BAD

```
foldr :: Foldable t
      ➔ (a → b → b)
      → b
      → (t a → b)
```

Qualified type
==
GOOD

```
find :: Foldable t ➔ (a → Bool)
     → t a → Maybe a
```

```
class Foldable t where
{
    fold :: Monoid m ➔ t m → m
    foldMap :: Monoid m ➔ (a → m) → t a → m
    foldr :: (a → b → b) → b → t a → b
    foldl :: (a → b → a) → a → t b → a
}

class Monoid m where
{
    mempty :: m
    mappend :: m → m → m
    mconcat :: [m] → m
}
```

```
fold :: Monoid m → t m → m
```

Implicit parameter
Controlled “injection”
Competent

Roughly
a shorthand for

```
fold :: (m, m→m→m,[m]→m)
      → t m
      → m
```

All dependencies are explicit
No “injection”
Dreyfus Novice

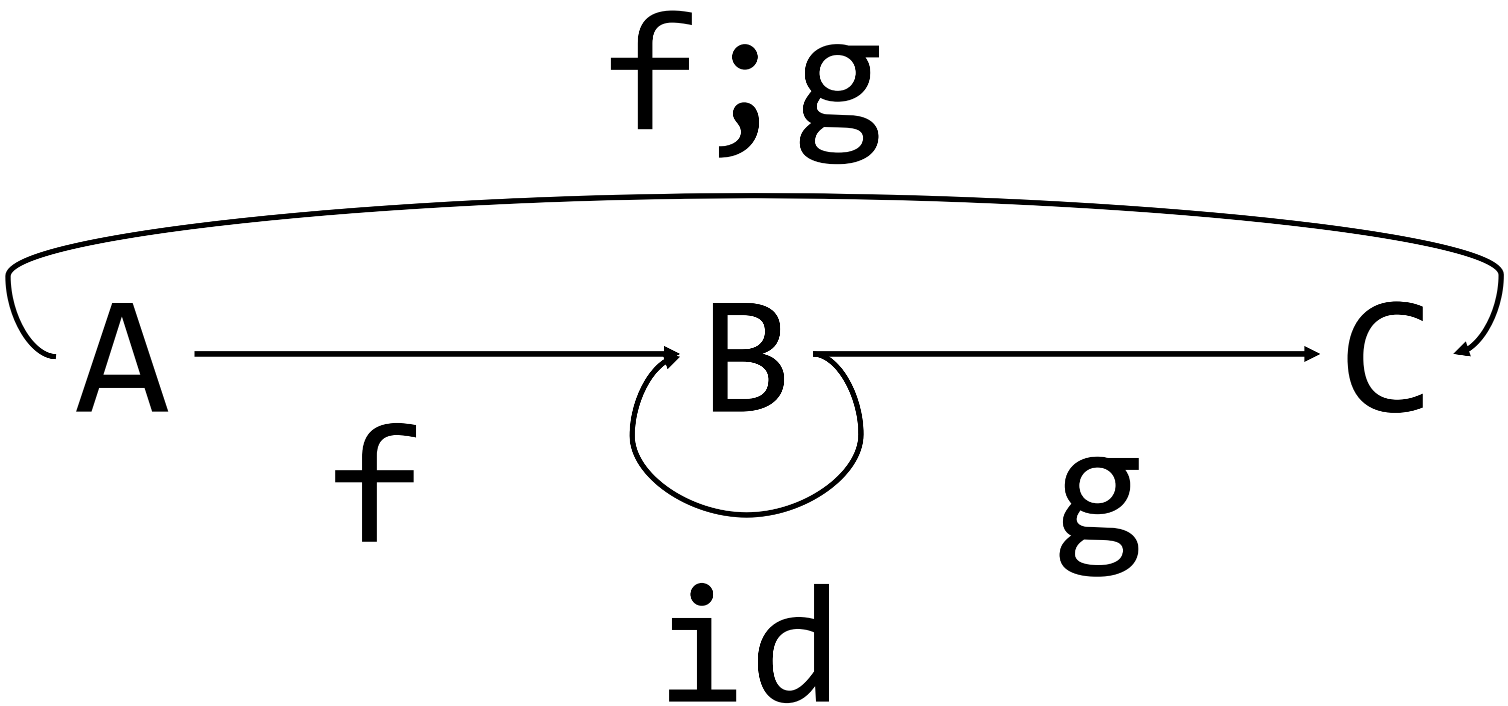
```
fold :: (m, m→m→m,[m]→m)
      → t m
      → m
```

Concrete type

==

BAD ☺

Interface-based programming to the extreme: Category Theory



Objects, Morphisms and Composition

Dreyfus proficient

Dual (category theory)

From Wikipedia, the free encyclopedia

In category theory, a branch of mathematics, **duality** is a correspondence between properties of a category C and so-called **dual properties** of the opposite category C^{op}. Given a statement regarding the category C, by interchanging the source and target of each morphism as well as interchanging the order of composing two morphisms, a corresponding dual statement is obtained regarding the opposite category C^{op}. **Duality**, as such, is the assertion that truth is invariant under this operation on statements. In other words, if a statement is true about C, then its dual statement is true about C^{op}. Also, if a statement is false about C, then its dual has to be false about C^{op}.

Given a concrete category C, it is often the case that the opposite category C^{op} per se is abstract. C^{op} need not be a category that arises from mathematical practice. In this case, another category D is also termed to be in **duality** with C if D and C^{op} are equivalent as categories.

In the case when C and its opposite C^{op} are equivalent, such a category is **self-dual**.

Obsession with monads *is* a medical condition (thanks Pat Helland)

If F and G are a pair of adjoint functors, with F left adjoint to G, then the composition $G \circ F$ is a monad. Therefore, a monad is an endofunctor. If F and G are inverse functors the corresponding monad is the identity functor. In general adjunctions are not equivalences — they relate categories of different natures. The monad theory matters as part of the effort to capture what it is that adjunctions 'preserve'. The other half of the theory, of what can be learned likewise from consideration of $F \circ G$, is discussed under the dual theory of **comonads**.

The monad axioms can be seen at work in a simple example: let G be the forgetful functor from the category **Grp** of groups to the category **Set** of sets. Then as F we can take the free group functor.

This means that the monad

$$T = G \circ F$$

takes a set X and returns the underlying set of the free group $\mathrm{Free}(X)$. In this situation, we are given two natural morphisms:

$$X \to T(X)$$

by including any set X in $\mathrm{Free}(X)$ in the natural way, as strings of length 1. Further,

$$T(T(X)) \to T(X)$$

can be made out of a natural concatenation or 'flattening' of 'strings of strings'. This amounts to two natural transformations

$$I \to T$$

and

$$T \circ T \to T$$

They will satisfy some axioms about identity and associativity that result from the adjunction properties.

Those axioms are formally similar to the monoid axioms. They are taken as the definition of a general monad (not assumed *a priori* to be connected to an adjunction) on a category.

Blah, blah, blah
Blah, blah
Blah, blah!

IEnumerable<T>

() → (()→T)

IEnumerator<T>

IObservable<T>

(T→()) → ()

Iobserver<T>

Monad
CoMonad
Dual?

Does it really matter …

Dreyfus expert

Functional "Programming" is a tool for thought

"Imperative" Programming is a tool for hacking

You are the Chef ….

```
T StarAnise(){ …}

Lazy<T> Cloves(){…}

Task<T> Cinnamon() {….}

IEnumerable<T> Pepper(){ …}

IObservable<T> Fennel() {…}
```