# C# is a functional programming language

Andrew Kennedy
Microsoft Research Cambridge

# Quicksort revisited

- Name the language...

C# 3.0

```
Func<intlist, intlist> Sort =
  xs =>
    xs.Case(
      () => xs,
      (head,tail) => (Sort(tail.Where(x => x < head)))
                       .Concat
                     (Single(head))
                       .Concat
                     (Sort(tail.Where(x => x >= head)))
    );
```

parameterized type of functions

higher-order function

lambda expression

append

type inference

recursion

filter

# The gap narrows...

- C# 3.0 has many features well-known to functional programmers
  - Parameterized types and polymorphic functions (*generics*)
  - First-class functions (*delegates*)
  - Lightweight lambda expressions & closure conversion
  - Type inference (for locals and lambdas)
  - Streams (*iterators*)
  - A library of higher-order functions for collections & iterators
  - And even: GADTs (*polymorphic inheritance*)
- This talk: is it serious competition for ML and Haskell?
  - (Note: Java 5 has many but not all of the above features)

# A brief history of fun in C#

- C# 1.0:
  - First-class functions (*delegates*), created only from named methods. Environment=object, code=method.
- C# 2.0:
  - Parameterized types and polymorphic methods (*generics*)
  - *Anonymous methods:* creation of delegate objects from code bodies, closure-converted by C# compiler
  - *Iterators*: stream abstraction, like *generators* from Clu
- C# 3.0:
  - *Lambda expressions:* lightweight syntax for anonymous methods whose bodies are expressions
  - *Type inference* for locals and lambdas
  - (Also, not discussed: expression trees for lambdas)

# Delegates (C# 1.0)

- Essentially *named* function types e.g.

```
delegate bool IntPred(int x);
```

- Delegate objects capture a method code pointer together with an object reference e.g.

```
class Point {
  int x; int y;
  bool Above(int ybound)
    { return y >= ybound; }
}
Point point;
IntPred predicate = new IntPred(point.Above);
```

- Compare (environment, code pointer) closure in a functional language.

# Generics (C# 2.0)

- Types (classes, interfaces, structs and delegates) can be parameterized on other types e.g.

```
delegate R Func<A,R>(A arg);
class List<T> { ... }
class Dict<K,D> { ... }
```

- Methods (instance and static) can be parameterized on types e.g.

```
static void Sort<T>(T[] arr);
static void Swap<T>(ref T x, ref T y);
class List<T> {
  List<Pair<T,U>> Zip<U>(List<U> other) ..
```

- Very few restrictions:
  - Parameterization over primitive types, reference types, structs
  - Types preserved at runtime, in spirit of the .NET object model

# Generics: expressiveness

1. Polymorphic recursion e.g.

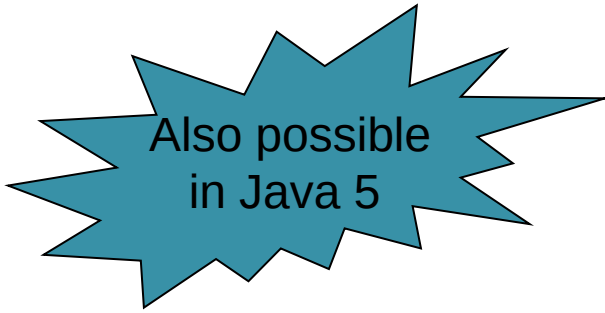
```
static void Foo<T>(List<T> xs) {
  …Foo<List<List<T>>>(…)…  }
```

2. First-class polymorphism (System F) e.g.

```
interface Sorter { void Sort<T>(T[] arr); }
class QuickSort : Sorter { … }
class MergeSort : Sorter { … }
```

3. GADTs e.g.

```
abstract class Expr<T> { T Eval(); }
class Lit : Expr<int> { int Eval() { … } }
class PairExpr<A,B> : Expr<Pair<A,B>>
  { Expr<A> e1; Expr<B> e2; Pair<A,B> Eval() { … } }
```



# Anonymous methods (C# 2.0)

- Delegates are clumsy: programmer has to name the function and "closure-convert" by hand
- So C# 2.0 introduced anonymous methods
  - No name
  - Compiler does closure-conversion, creating a class and object that captures the environment e.g.

```
bool b = xs.Exists(delegate(int x) { return x>y; });
```

Local y is free in body of anonymous method

# IEnumerable<T>

- Like Java, C# provides interfaces that abstract the ability to enumerate a collection:

```
interface IEnumerable<T>
{ IEnumerator<T> GetEnumerator(); }
interface IEnumerator<T> {
  T Current { get; }
  bool MoveNext();
}
```

- To "consume" an enumerable collection, we can use the foreach construct:

```
foreach (int x in xs) { Console.WriteLine(x); }
```

- But in C# 1.0, implementing the "producer" side was error-prone (must implement Current and MoveNext methods)

# Iterators (C# 2.0)

- C# 2.0 introduces *iterators*, easing task of implementing IEnumerable e.g.

```
static IEnumerable<int> UpAndDown(int bottom, int top) {
  for (int i = bottom; i < top; i++) { yield return i; }
  for (int j = top; j >= bottom; j--) { yield return j; }
}
```

- Iterators can mimic functional-style streams. They can be infinite:

```
static IEnumerable<int> Evens() {
  for (int i = 0; true; i += 2) { yield return i; } }
```

- The System.Query library provides higher-order functions on IEnumerable<T> for map, filter, fold, append, drop, take, etc.

```
static IEnumerable<T> Drop(IEnumerable<T> xs, int n) {
  foreach(T x in xs) { if (n>0) n--; else yield return x; }}
```

# Lambda expressions

- Anonymous methods are just a little too heavy compared with lambdas in Haskell or ML: compare

```
delegate (int x, int y) { return x*x + y*y; }
\(x,y) -> x*x + y*y
fn (x,y) => x*x + y*y
```

- C# 3.0 introduces *lambda expressions* with a lighter syntax, inference (sometimes) of argument types, and expression bodies:

```
(x,y) => x*x + y*y
```

- Language specification simply defines lambdas by translation to anonymous methods.

# Type inference (C# 3.0)

- Introduction of generics in C# 2.0, and absence of type aliases, leads to type*full* programs!

```
Dict<string,Func<int,Set<int>>> d = new
Dict<string,Func<int,Set<int>>>();
Func<int,int,int> f = delegate (int x, int y) { return x*x + y*y; }
```

- C# 3.0 supports a modicum of type inference for local variables and lambda arguments:

```
var d = new Dict<string,Func<int,Set<int>>>();
Func<int,int,int> f = (x,y) => x*x + y*y;
```

# GADTs

- Generalized Algebraic Data Types permit constructors to return *different* instantiations of the defined type
- Canonical example is well-typed expressions e.g.

```
datatype Expr a with
  Lit : int      Expr int
| PairExpr : Expr a      Expr b      Expr (a × b)
| Fst : Expr (a × b)      Expr a …
```

- In C#, we can render this using "polymorphic inheritance":

```
abstract class Expr<a>
class Lit : Expr<int> { int val; … }
class PairExpr<a,b> : Expr<Pair<a,b>> { Expr<a> e1; Expr<b> e2; … }
class Fst<a,b> : Expr<a> { Expr<Pair<a,b>> e; … }
```

- *Demo*: strongly-typed printf

# Implementation

- C# is compiled to IL, an Intermediate Language that is executed on the .NET Common Language Runtime
- The CLR has direct support for many of the features described here
  - Delegates are special classes with fast calling convention
  - Generics (parametric polymorphism) is implemented by just-in-time specialization so *no boxing* is required
  - Closure conversion is done by the C# compiler, which shares environments between closures where possible

# Putting it together

1. Take your favourite functional pearl
2. Render it in C# 3.0

- Here, Hutton & Meijer's monadic parser combinators. *Demo.*

# Fun in C#: serious competition?

- It's functional programming bolted onto a determinedly imperative object-oriented language
  - Quite nicely done, but C# 3.0 shows its history
  - The additional features in C# 3.0 were driven by the LINQ project (Language INtegrated Query)
- Contrast Scala, which started with (almost) a clean slate:
  - Object-oriented programming (new design) + functional programming (new design)
- Many features remain the preserve of functional languages
  - Datatypes & pattern matching
  - Higher-kinded types, existentials, sophisticated modules
  - Unification/constraint-based type inference
  - True laziness

# Closures might surprise you...

- Guess the output

```
var funs = new Func<int,int>[5]; // Array of functions of type int    int
for (int i = 0; i<5; i++)
{
   funs[i] = j => i+j;  // To position index i, assign    j. i+j
}
Console.WriteLine(funs[1](2));
```

Result is “7”!

- Why? Clue: r-values vs l-values. Arguably, the right design:

```
static void While(VoidFunc<bool> condition, VoidFunc action) { … }
int x = 1;While(() => x < 10, () => { x=2*x; });
```

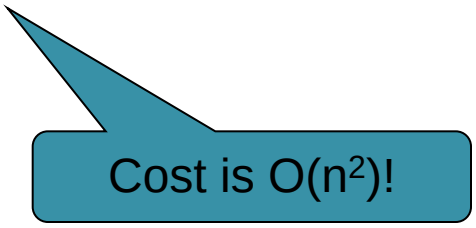
# Iterators might surprise you…

- Iterator combinators can be defined purely using foreach and yield.

```
X Head<X>(IEnumerable<X> xs)
  {  foreach (X x in xs) { return x; } }
IEnumerable<X> Tail<X>(IEnumerable<X> xs)
  { bool head = true;
    foreach (X x in xs) { if (head) head = false; else yield return x; } }
```

- But performance implications are surprising:

```
IEnumerable<int> xs;
for (int i = 0; i < n; i++) { xs = Tail(xs); }
int v = Head(xs);
```



# Performance

- Closure creation and application are relatively cheap operations
  - But almost no optimizations are performed. Contrast ML/Haskell uncurrying, arity-raising, flow analysis, etc.
- Iterators are *not* lazy streams
  - No memoizing of results
  - Chaining of IEnumerable wrappers can lead to worsening of asymptotic complexity
  - Though there's nothing to prevent the programmer implementing a proper streams library, as in ML

# Try it yourself

- C# 2.0 is part of .NET Framework 2.0 SDK available free from http://msdn.microsoft.com/downloads/
- Also Visual C# Express Edition: a free lightweight version of Visual Studio http://msdn.microsoft.com/vstudio/express/visualcsharp/
- Download preview of C# 3.0 from http://msdn.microsoft.com/data/ref/linq/