The F# Programming Language and Scientific Computing

Andrew Kennedy
Researcher
Microsoft Research Cambridge
What is F#?

• It’s a **new** programming language from Microsoft, in Visual Studio 2010 alongside established languages C++, C# and Visual Basic
• It’s takes some **old** ideas from two decades of research in *functional programming* (SML, Haskell, Caml, Erlang, Clean, Scheme)...
• ... and adds some interesting **new** twists
  • Smooth integration with .NET
  • Asynchronous computation, supporting parallelism, reactive programming
  • Active patterns, supporting extensible pattern matching
  • Units-of-measure types for scientific computation
  • Code quotation, for meta-programming
  • ...
The Cambridge connection

C# Generics invented here (Don Syme, Andrew Kennedy)

Home of Haskell (Simon Peyton Jones, et al)

Hotbed of biology (Luca Cardelli, et al)

Birthplace of F# (Don Syme)
What is functional programming?

- Mainstream *imperative* programming is all about *state*, and *update*
- *Object-oriented* programming is its modern incarnation

```csharp
static int Sum(int[] xs) {
    int result = 0;
    foreach (int x in xs) { result += x; }
    return result;
}
```

- Functional programming is all about *values* and *functions* without side-effects. It enables a Lego-like approach to programming, combining simple building blocks using functions:

```fsharp
let sum xs = List.fold (+) 0 xs
```

Updating the state

fold takes the function `'+'` as parameter
Two flavours of functional programming

**Pure**

No side-effects at all: values are immutable, no state
Mathematics has got on just fine with pure functions for centuries!
Example languages: Haskell, Clean

**Impure**

Some values are mutable, functions can have side-effects
Example languages: Standard ML, OCaml, Scheme, Erlang
F# is assuredly in the *impure* camp:

**Similar core language**

OCaml

F#

C#/.NET

**Similar object model**
Why no one uses functional languages

Editor: Philip Wadler, Bell Laboratories, Lucent Technologies, wadler@research.bell-labs.com

Philip Wadler

To say that no one uses functional languages is an exaggeration. Phone calls in the European Parliament are routed by programs written in Ericsson’s functional language Erlang. Virtual CDs are distributed on Cornell’s network via the Ensemble system written in INRIA’s CAML, and real CDs are shipped by Polygram in Europe using Software AG’s Natural Expert. Functional languages are the language of choice for writing theorem provers, including the HOL system which helped debug the design of the HP 9000 line of multiprocessors. These applications and others are described in a previous column [1].

Still … I work at Bell Labs, where C and C++ were invented. Compared to users of C, “no one” is a tolerably accurate count of the users of functional languages.

Advocates of functional languages claim they produce an order of magnitude improvement in productivity. Experiments don’t always verify that figure — sometimes they show an improvement of only a factor of four. Still, code that’s four times as short, four times as quick to write, or four times easier to maintain is not to be sniffed at. So why aren’t functional languages more widely used?

rather than built from scratch. Many of these components are written in C or C++, so a foreign function interface to C is essential, and interfaces to other languages can be useful.

The isolationist nature of functional languages is beginning to give way to a spirit of open interchange. Serious implementations now routinely provide interfaces to C, and sometimes other languages. Interworking with the imperative world is straightforward for strict languages like ML or Erlang, but trickier for lazy languages like Haskell or Clean, since laziness makes the order of evaluation difficult to predict. However, through clever interplay of theory and practice, recent research has shown how abstract concepts such as monads and side effects can be applied to smoothly interface lazy functional languages to the real world [2, 3].

Conquering isolationism is a task for everyone, not just functional programmers. The computing industry is now beginning to deploy standards, such as CORBA and COM, that support the construction of software from reusable components. Recent work allows any Haskell program to be packaged as a COM component, and any COM component to be called from Haskell. Among other applications, this allows Haskell to be used as a scripting language for Microsoft’s Internet Explorer web browser.

1 Reasons

F# smoothly integrates .NET

Written in 1998
Functional programming in 2010

• Since 1998, mainstream languages have borrowed features and ideas from f.p. languages. Many research success stories!
  • Parametric polymorphism (generics in Java, C#, Visual Basic)
  • First-class functions (closures in Python, lambdas in C#, soon in C++)
  • Type inference (inferred generics in Java, the var keyword in C#)
  • Pattern matching (now in Ruby)

• So: I can do functional programming in C#. Why use F#?
  • Because it’s “functional-first”, and features were designed in tandem, not bolted onto an existing language.
  • This leads to a great simplicity. See what you think...
What are functional languages used for?

Traditionally:

To implement compilers for functional languages 😊

Actually, much more. Already, F# is used in

- Financial analysis
- Data mining
- Machine learning
- Bioinformatics
- Web programming

... and to implement compilers for functional languages
Remainder of talk

• The simplicity of functional programming in F#
• A taste of F#: demos in Visual Studio
• Feature focus: Units of Measure (see also at DemoFest)
• Wrap up and Q&A
Simplicity: functions as values

**Pleasure**

Function type

F#

type Command = Command of (Rover -> unit)

let BreakCommand = Command(fun rover -> rover.Accelerate(-1.0))

let TurnLeftCommand = Command(fun rover -> rover.Rotate(-5.0<degs>))

**Pain**

C#

abstract class Command
{
    public virtual void Execute();
}

abstract class MarsRoverCommand : Command
{
    protected MarsRover Rover { get; private set; }
    public MarsRoverCommand(MarsRover rover)
    {
        this.Rover = rover;
    }
}

class BreakCommand : MarsRoverCommand
{
    public BreakCommand(MarsRover rover) : base(rover) { }
    public override void Execute()
    {
        Rover.Rotate(-5.0);
    }
}

class TurnLeftCommand : MarsRoverCommand
{
    public TurnLeftCommand(MarsRover rover) : base(rover) { }
    public override void Execute()
    {
        Rover.Rotate(-5.0);
    }
}
Simplicity: tuples

let swap (x, y) = (y, x)

Pleasure

let rotations (x, y, z) =
[ (x, y, z);
  (z, x, y);
  (y, z, x) ]

Tuples

let reduce f (x, y, z) =
  f x + f y + f z

Tuplet pattern

Pain

# Tuple<"U","T"> Swap("T","U")

Tuple<"U","T"> Swap<"T","U">(Tuple<"T","U"> t)
{
  return new Tuple<"U","T">(t.Item2, t.Item1)
}

# ReadOnlyCollection<Tuple<"T","T","T">>

ReadOnlyCollection<Tuple<"T","T","T">>
Rotations<"T">(Tuple<"T","T","T"> t)
{
  new ReadOnlyCollection<int>
  (new Tuple<"T","T","T">[
    new Tuple<"T","T","T">(t.Item1, t.Item2, t.Item3);
    new Tuple<"T","T","T">(t.Item3, t.Item1, t.Item2);
    new Tuple<"T","T","T">(t.Item2, t.Item3, t.Item1);
  ]);}

# int Reduce<"T">(Func<"T",int> f,Tuple<"T","T","T"> t)

int Reduce<"T">(Func<"T",int> f,Tuple<"T","T","T"> t)
{
  return f(t.Item1)+f(t.Item2)+f(t.Item3);
}
public abstract class Expr {}
public abstract class UnaryOp : Expr
{
    public Expr First { get; private set; }
    public UnaryOp(Expr first)
    {
        this.First = first;
    }
}

public abstract class BinExpr : Expr
{
    public Expr First { get; private set; }
    public Expr Second { get; private set; }
    public BinExpr(Expr first, Expr second)
    {
        this.First = first;
        this.Second = second;
    }
}
public class TrueExpr : Expr {}

public class And : BinExpr
{
    public And(Expr first, Expr second) : base(first, second) { }
}
public class Nand : BinExpr
{
    public Nand(Expr first, Expr second) : base(first, second) { }
}
public class Or : BinExpr
{
    public Or(Expr first, Expr second) : base(first, second) { }
}
public class Xor : BinExpr
{
    public Xor(Expr first, Expr second) : base(first, second) { }
}
public class Not : UnaryOp
{
    public Not(Expr first) : base(first) { }
}

Simplicity: types for abstract syntax

type Expr =
| True
| And of Expr * Expr
| Nand of Expr * Expr
| Or of Expr * Expr
| Xor of Expr * Expr
| Not of Expr

Pleasure

Recursive discrimination type
public abstract class Expr {
    public abstract bool Eval();
}
public abstract class UnaryOp : Expr {
    public Expr First { get; private set; }
    public UnaryOp(Expr first) {
        this.First = first;
    }
}
public abstract class BinExpr : Expr {
    public Expr First { get; private set; }
    public Expr Second { get; private set; }
    public BinExpr(Expr first, Expr second) {
        this.First = first;
        this.Second = second;
    }
}
public class TrueExpr : Expr {
}
public class And : BinExpr {
    public And(Expr first, Expr second) : base(first, second) {
        bool Evalu() { return First.Eval() && Second.Eval(); }
    }
}
public class Nand : BinExpr {
    public Nand(Expr first, Expr second) : base(first, second) {
    }
}
public class Or : BinExpr {
    public Or(Expr first, Expr second) : base(first, second) {
        bool Evalu() { return First.Eval() || Second.Eval(); }
    }
}
public class Xor : BinExpr {
    public Xor(Expr first, Expr second) : base(first, second) {
    }
}
public class Not : UnaryOp {
    public Not(Expr first) : base(first) {
        bool Evalu() { return !First.Eval(); }
    }
}

Simplicity: pattern matching

### Pain

```
let eval e =
    match e with
    | True -> true
    | And (e1, e2) -> eval e1 && eval e2
    | Or (e1, e2) -> eval e1 || eval e2
    | Not e -> not (eval e)
    ...
```

### Pleasure

```
public abstract class Expr {
    public abstract bool Eval();
}
public abstract class UnaryOp : Expr {
    public Expr First { get; private set; }
    public UnaryOp(Expr first) {
        this.First = first;
    }
}
public abstract class BinExpr : Expr {
    public Expr First { get; private set; }
    public Expr Second { get; private set; }
    public BinExpr(Expr first, Expr second) {
        this.First = first;
        this.Second = second;
    }
}
public class TrueExpr : Expr {
}
public class And : BinExpr {
    public And(Expr first, Expr second) : base(first, second) {
        bool Evalu() { return First.Eval() && Second.Eval(); }
    }
}
public class Nand : BinExpr {
    public Nand(Expr first, Expr second) : base(first, second) {
    }
}
public class Or : BinExpr {
    public Or(Expr first, Expr second) : base(first, second) {
        bool Evalu() { return First.Eval() || Second.Eval(); }
    }
}
public class Xor : BinExpr {
    public Xor(Expr first, Expr second) : base(first, second) {
    }
}
public class Not : UnaryOp {
    public Not(Expr first) : base(first) {
        bool Evalu() { return !First.Eval(); }
    }
}
Simplicity: objects

Pleasure

type Vector2D(dx:double, dy:double) =

  member v.DX = dx
  member v.DY = dy
  member v.Length = sqrt(dx*dx+dy*dy)
  member v.Scale(k) = Vector2D(dx*k,dy*k)

“F# as a better C#”

Pain

class Vector2D {
  private double dx;
  private double dy;
  public Vector2D(double dx, double dy) {
    this.dx = dx; this.dy = dy; }
  public double DX { get { return dx; } }
  public double DY { get { return dy; } }
  public double Length {
    get { return Math.Sqrt(dx*dx + dy*dy); }
  }
  public Vector2D Scale(double k) {
    return new Vector2D(dx*k, dy*k); }
}
Simplicity: parallelism

- Functional programs are ideally suited to parallelization
  - Data is immutable
  - Functions are pure (non-side-effecting)

F#'s `async` feature combines with .NET support for parallelism:

```fsharp
Async.Parallel [ http "www.google.com";
                 http "www.bing.com";
                 http "www.yahoo.com"; ]
```

```fsharp
Async.Parallel [ for i in 0 .. 200 -> computeTask i ]
```
Demo 1: Biology

- Andrew Phillips from MSR Cambridge:

  A programming language for composable DNA circuits

  Recently, a range of information-processing circuits have been

- F# implementation
  - Core is purely-functional
  - GUI is object-oriented
Demo 2: Physics

• Simulation of pole balancing on trolley (classic control theory)
• Implemented by a Philip Hennig, physics PhD student, interning at MSR - his first experience of F#
• Object-oriented F# with mutation: *not* purely functional!
• Nice demonstration of units of measure
Q: Is F# a scripting language?
A: It depends what you mean by *scripting language*. It has the succinct syntax and absence of type information common in scripting languages. Yet it is statically type-checked and compiled!
Q: Scripting languages suffer from poor performance. What about F#?
A: See above: it’s a compiled, statically-type-checked language, and performance is good. (e.g. see web for favourable comparison of matrix performance with MATLAB and C++.)
Metric mishap caused loss of NASA orbiter

September 30, 1999
Web posted at: 4:21 p.m. EDT (2021 GMT)

In this story:

- Metric system used by NASA for many years
- Error points to nation’s conversion lag

Related Stories, Sites

NASA’s Climate Orbiter was lost September 23, 1999

By Robin Lloyd
CNN Interactive Senior Writer

(CNN) -- NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency’s team used the more conventional metric system for a key spacecraft operation, according to a review finding released Thursday.

The units mismatch prevented navigation information from transferring between the Mars Climate Orbiter spacecraft team in at Lockheed Martin in Denver and the flight team at NASA’s Jet Propulsion Laboratory in Pasadena, California.
What went wrong?

- NASA’s $125 million Mars Climate Orbiter was lost
- Software authored by NASA used *newtons* (N) for units of force
- Software authored by its contractor used *pound-force* (lbf) instead
- Result: the wrong force was applied to the thrusters, taking the spacecraft into an orbit that was too low.

Crash!
Type checking for units?

- Many programming languages do static type-checking, preventing errors such as adding integers to strings:

  ```csharp
  int age = 17;
  string name = "Joe";
  int silly = age + name;
  ```

  ```csharp
  Cannot implicitly convert type 'string' to 'int'
  ```

- Units are the “types” of science. But in most programming languages numbers are just numbers, without units. Not in F#!

  ```fsharp
  let height = 1.85<m>
  let weight = 70.0<kg>
  let x = height + weight
  ```

  ```fsharp
  The unit of measure 'kg' does not match the unit of measure 'm'
  ```
Demo 3: Physics in games

• Lunar lander game written by games programmer Phillip Trelford
• Object-oriented style, lots of units of measure
• Uses the XNA framework (Xbox framework)
Three steps to unit heaven

1. Declare base and derived units

```fsharp
[<Measure>] type kg       // kilogram
[<Measure>] type m        // metre
[<Measure>] type s        // second
[<Measure>] type N = kg m / s^2 // newton
[<Measure>] type J = N m   // joule
```

2. Annotate constants with units

```fsharp
let g = 9.81<m/s^2>       // acceleration due to gravity
```

3. F# infers units everywhere else, and reports errors

```fsharp
// Find the mean, variance and standard deviation
let mean(xs) = sum(xs) / float(xs.Length)
let variance(xs) = mean(Array.map sqr xs) - sqr(mean(xs))
let variance : float<'u> array -> float<'u^2>
```

// Full name: Examples.variance
Generics and units

• Functions can be generic over units

```
let sqr (x:float<_>) = x*x
```

```
val sqr:float<'u> -> float<'u ^ 2>
Full name: Units2.sqr
```

• Data types can be parameterized on units

```
type Complex[<Measure>] 'u' (re:float<'u>, im:float<'u>) =
  member this.re : float<'u> = re
  member this.im : float<'u> = im
static member (+) (a:Complex<'u>,b:Complex<'u>) = new Complex<'u>(a.re + b.re, a.im + b.im)
static member (-) (a:Complex<'u>,b:Complex<'u>) = new Complex<'u>(a.re - b.re, a.im - b.im)
```
How does it work?

- Q: This idea seems obvious enough. Has it been done before?
- A: It’s been proposed many times, but only implemented in research languages. And inferring units isn’t that easy. (It was the subject of my PhD thesis!)

- Q: How’s it done?
- A: When F# doesn’t know the units, it leaves a “placeholder” variable. When it detects that units must match up (e.g. in an addition), it generates an equation between units. It then solves the equations by Gaussian elimination. Lots more details, invisible to the programmer.
Want to know more about F#?

- Try it!
  - Get Visual Studio 2010
  - Or: download the free “Visual Studio 2008 shell”, and install the April 2010 F# Community Technical Preview
  - Or: install Mono on Linux, and the F# CTP
- Interviews, talks, tutorials, etc. Start at http://fsharp.net
- Blogs:
  http://blogs.msdn.com/andrewkennedy
  http://blogs.msdn.com/dsyme