FSharp.Core.dll Specification

[FSharp.Core.dll Specification 1](#_Toc265492351)

[1 Overview 4](#_Toc265492352)

[1.1 Compiled Binaries 4](#_Toc265492353)

[1.2 Design Influences 4](#_Toc265492355)

[1.3 Naming 5](#_Toc265492356)

[1.4 Design Goals and User Experience 5](#_Toc265492359)

[1.4.1 Design Goals: Minimalist Application Development Scenarios 5](#_Toc265492360)

[1.4.2 Design Goals: Versioning and Deployment 5](#_Toc265492361)

[1.4.3 Design Goal: Role in Minimalist Component Development Scenarios 5](#_Toc265492362)

[1.4.4 Design Goal: .NET 2.0 – 4.0 binary compatibility 5](#_Toc265492363)

[1.4.5 Design Goal: Transparent Security 5](#_Toc265492364)

[1.4.6 Design Goal: Not a General Purpose .NET DLL 5](#_Toc265492365)

[1.4.7 Design Goal: A Good .NET Citizen (Alignment and Sharing Considerations) 6](#_Toc265492366)

[1.4.8 Design Goal: One Way to do Things 6](#_Toc265492367)

[1.4.9 Design Goal: Enable a F# -style Functional Programming Language Subset 7](#_Toc265492368)

[1.4.10 Design goal: Binary Size 7](#_Toc265492369)

[1.4.11 Design goal: Compiled Names follow .NET Standards 7](#_Toc265492370)

[1.5 Non-design goals 7](#_Toc265492371)

[1.5.1 Non-design goal: OCaml Compatibility 7](#_Toc265492372)

[1.5.2 Non-design goal: Full Backwards Compatibility with Early F# 7](#_Toc265492373)

[1.5.3 Non-design goal: FxCop Perfection 7](#_Toc265492375)

[1.5.4 Non-design goal: Minimalism 8](#_Toc265492376)

[1.6 Historical API Design Work 8](#_Toc265492377)

[2 Design Guidelines 9](#_Toc265492378)

[2.1 Standard F# Operators 9](#_Toc265492379)

[2.2 Use of Tuples 10](#_Toc265492380)

[2.3 Use of F# Function Values 10](#_Toc265492381)

[2.4 Currying Guidelines 11](#_Toc265492382)

[2.5 Overloading Guidelines and Type Inference 11](#_Toc265492383)

[2.6 Capitalization Guidelines 11](#_Toc265492384)

[2.7 Underscore Guidelines 12](#_Toc265492385)

[2.8 Abbreviation Guidelines 12](#_Toc265492390)

[3 FSharp.Core – Basic Operators 13](#_Toc265492391)

[3.1 Basic Type Abbreviations 13](#_Toc265492392)

[3.2 Basic Arithmetic Operators 13](#_Toc265492393)

[3.3 Generic Equality and Comparison Operators 13](#_Toc265492394)

[3.4 Bitwise manipulation operators 13](#_Toc265492395)

[3.5 Math operators 14](#_Toc265492396)

[3.6 Function Pipelining and Composition Operators 14](#_Toc265492397)

[3.7 Object Transformation Operators 14](#_Toc265492398)

[3.8 Pair Operators 15](#_Toc265492399)

[3.9 Exception Operators 15](#_Toc265492400)

[3.10 Input/Output Handles 15](#_Toc265492401)

[3.11 Overloaded Conversion Functions 15](#_Toc265492402)

[3.12 Checked Arithmetic Operators 16](#_Toc265492403)

[3.13 Type Abbreviations 16](#_Toc265492404)

[3.14 Special Types 16](#_Toc265492405)

[3.14.1 byref<\_> 16](#_Toc265492406)

[3.14.2 float<\_>, float32<\_>, decimal<\_> 16](#_Toc265492407)

[3.14.3 nativeptr<\_> 16](#_Toc265492408)

[3.15 FSharp.Core.MatchFailure (Exception) 17](#_Toc265492464)

[3.16 Attribute and Attribute Flag Types 17](#_Toc265492465)

[4 FSharp.Core - Basic Types 18](#_Toc265492466)

[4.1 FSharp.Core.Ref<T> (Type, compiled name FSharpRef<T>) 18](#_Toc265492477)

[4.2 FSharp.Core.Option<T> (Type+Module) 18](#_Toc265492478)

[4.3 FSharp.Core.String (Module) 20](#_Toc265492479)

[5 FSharp.Collections (Namespace) 21](#_Toc265492481)

[5.1 Design Considerations 21](#_Toc265492482)

[5.2 FSharp.Collections.List (Type+Module) 21](#_Toc265492483)

[5.3 FSharp.Collections.Array (Module) 24](#_Toc265492484)

[5.4 FSharp.Collections.Array.Parallel (Module) 25](#_Toc265492489)

[5.5 FSharp.Collections.Seq (Module) 27](#_Toc265492490)

[5.6 FSharp.Collections.Set (Type+Module) 28](#_Toc265492495)

[5.7 FSharp.Collections.Map<Key,Value> (Type) 31](#_Toc265492496)

[5.8 FSharp.Collections.Array2D (Module) 32](#_Toc265492499)

[5.9 FSharp.Collections.Array3D (Module) 33](#_Toc265492500)

[5.10 FSharp.Collections.Array4D (Module) 33](#_Toc265492501)

[5.11 Convenience Functions (FSharp.Core.ExtraTopLevelOperators Module, AutoOpen) 33](#_Toc265492502)

[5.11.1 Collection Functions 33](#_Toc265492503)

[5.11.2 Printing Functions (FSharp.Core.ExtraTopLevelOperators Module) 33](#_Toc265492504)

[6 FSharp.Control – Events and Observables 35](#_Toc265492505)

[7 FSharp.Control – Asynchronous Programming 37](#_Toc265492506)

[7.1 Dependency Basis 38](#_Toc265492507)

[7.2 Execution Semantics 38](#_Toc265492508)

[7.2.1 Threaded Context 38](#_Toc265492509)

[7.2.2 Cancellation 38](#_Toc265492510)

[7.2.3 Timeouts 39](#_Toc265492511)

[7.2.4 Return to Synchronization Context 39](#_Toc265492512)

[7.2.5 Serialization and Cross-Machine/Cross-AppDomain programming: 39](#_Toc265492513)

[7.3 FSharp.Control.Async<T> 40](#_Toc265492515)

[7.4 FSharp.Control. AsyncBuilder 40](#_Toc265492516)

[7.5 FSharp.Control. Async 41](#_Toc265492517)

[7.5.1 Async.FromBeginEnd 42](#_Toc265492570)

[7.5.2 Async.AwaitEvent 43](#_Toc265492571)

[7.5.3 Async.AwaitWaitHandle 43](#_Toc265492572)

[7.5.4 Async.AwaitTask 43](#_Toc265492573)

[7.5.5 Async.AsBeginEnd 44](#_Toc265492574)

[7.5.6 Async.StartWithContinuations 44](#_Toc265492575)

[7.5.7 Async.StartChild 44](#_Toc265492576)

[7.6 FSharp.Control.CommonExtensions (Module, AutoOpen) 45](#_Toc265492577)

[7.7 FSharp.Control.WebExtensions (Module, AutoOpen) 45](#_Toc265492578)

[7.8 FSharp.Control.MailboxProcessor (Type) 46](#_Toc265492582)

[8 FSharp.Control – Lazy Programming 49](#_Toc265492583)

[8.1 FSharp.Control.Lazy (Type) 49](#_Toc265492584)

[9 FSharp.Reflection (Namespace) 50](#_Toc265492602)

[9.1 FSharp.Reflection.FSharpValue (Static Class) 50](#_Toc265492603)

[9.2 FSharp.Reflection.FSharpType and UnionCase (Static Class) 51](#_Toc265492604)

[10 FSharp.Quotations (Namespace) 52](#_Toc265492605)

[10.1 FSharp.Quotations.Var, Expr (Type) 52](#_Toc265492606)

[10.2 FSharp.Quotations.Patterns (Module) 53](#_Toc265492607)

[10.3 FSharp.Quotations.DerivedPatterns (Module) 54](#_Toc265492608)

[10.4 FSharp.Quotations.ExprShape (Module) 55](#_Toc265492609)

[11 FSharp.Text (Namespace) 56](#_Toc265492610)

[11.1 FSharp.Text.Format (Type) 56](#_Toc265492611)

[11.2 FSharp.Text.Printf (Module) 56](#_Toc265492612)

[12 FSharp.NativeInterop (Namespace) 58](#_Toc265492613)

[12.1 FSharp.NativeInterop.NativePtr (Module) 58](#_Toc265492614)

[13 Language Primitives 59](#_Toc265492618)

[13.1 FSharp.Core.FastFunc (Type) 59](#_Toc265492619)

[13.2 Tuple (functional tuples) 59](#_Toc265492620)

[13.3 FSharp.Core.TypeFunc (Type) 60](#_Toc265492623)

[13.4 FSharp.Core.Choice (Type) 60](#_Toc265492624)

[13.5 FSharp.Core.LanguagePrimitives (Module) 60](#_Toc265492625)

[13.6 FSharp.Core.OptimizedRanges (Module) 61](#_Toc265492626)

[13.7 FSharp.Core. OptimizedClosures (Module) 62](#_Toc265492627)

[13.8 FSharp.Core.Collecitons.SequenceExpressionHelpers (Module) 62](#_Toc265492628)

[13.9 FSharp.Collections.ComparisonIdentity (Module) 63](#_Toc265492629)

[13.10 FSharp.Collections.HashIdentity (Module) 63](#_Toc265492630)

[13.11 FSharp.Core.FuncConvert (Type) 63](#_Toc265492631)

[13.12 FSharp.Core.Unit (Type) 64](#_Toc265492632)

# Overview

FSharp.Core.dll is the standard library for F# applications, akin to Microsoft.VisualBasic.dll.  All F# applications and libraries must use it, and it contains

* F# operator definitions and runtime helper functions emitted by the compiler
* The library support for the F# functional programming model.
  + The basic functional programming types.
  + F# modules of functional programming operators to manipulate these.
  + The types are options, first class lazy values, functional lists, functional sets, functional maps and first class events.
* The library support for the F# asynchronous programming model.
* The library support for the F# quotation programming model.
* The library support for the F# reflection programming model.

## Compiled Binaries

This assembly is compiled as follows

* A supported .NET 2.0 version of this DLL, shipped as part of the F# redist SKU and the F# CTP
* A supported .NET 4.0 version of this DLL, shipped as part of the F# redist SKU and the F# CTP
* An unsupported Silverlight 3.0 version of this DLL, shipped as part of the F# CTP
* A supported Silverlight 4.0 version of this DLL, shipped as part of the Silverlight SDK
* An unsupported .NET CF 3.0 version of this DLL, shipped as part of the F# CTP, used particularly for XBox 360 programming
* Future: An unsupported .NET CF 3.7 version of this DLL, for WP7 programming

Some functionality is disabled on some of the unsupported older target platforms.

## Design Influences

This library is not a standard .NET library – it is highly tuned toward the F# development experience.  The design of the APIs in FSharp.Core.dll has been influenced by different factors:

* **Functional/F# Programming Style**:  The functional programming types and operators are in many aspects motivated by the F# language design itself.  F# tuples, lists, options, 1st class functions, type inference and pattern matching are all leveraged in these APIs, emphasizing the different style of programming that motivates F#’s existence, and makes it a productive tool in many domains.
* **1st class .NET citizen**: A major goal is to make F# “feel completely natural” on .NET.  This is sometimes in conflict with the bullet.
* **Compatibility with prior F# programming practice:**   While we’re still willing to take significant breaking changes (and do), F# has a non-trivial existing user base, as well as books and many code samples on the web, so this factor is relevant to API design discussions.
* **Setting the standard for F# API design:**  The design of these libraries inspires the design of much user-authored F# code.  We want to ensure that we practice what we preach for good F# design. See the F# Design Guidelines Document for details here.

## Naming

The name of this library is FSharp.Core.dll. After long consideration we followed the “Iron Python convention”, which considers the language to be the primary organization (c.f. IronPython.dll, IronPython.Math.dll). This recognizes that language communities act as organizations that transcend companies, and that this is useful to the long-term perception of a language.

It is feasible that the .NET naming conventions could now explicitly recognize this convention, since Iron Python has set an excellent standard here.

## Design Goals and User Experience

This section summarizes the design goals of this DLL.

### Design Goals: Minimalist Application Development Scenarios

It is a design goal that an F# development experience should be possible with just a functioning fsc.exe + FSharp.Core.dll + CLI.

### Design Goals: Versioning and Deployment

Ths library is designed for users with stringent stability, servicing and perf requirements on par with other .NET libraries

* We expect new “major update” version every ~2 years. Breaking changes very rare in these updates (if at all)
* Simple (or no) redistributable, and simple experience to chain the redistributable into an installation process

### Design Goal: Role in Minimalist Component Development Scenarios

It must be feasible to write low-dependency F# components for use from any .NET language. It must be possible to write F# and vanilla-.NET components that work with other .NET languages and yet rely only on FSharp.Core.dll

### Design Goal: .NET 2.0 – 4.0 binary compatibility

Binding redirect policy DLLs are used on .NET 4.0 to redirect references to this DLL to be references to the .NET 4.0 FSharp.Core DLL.

This means the .NET 2.0 version of this DLL must be 100% binary compatible with the .NET 4.0 version of this DLL.

### Design Goal: Transparent Security

The library should be security transparent.

### Design Goal: Not a General Purpose .NET DLL

The library is intended to be an F# library, not a .NET library

* A small subset may used regularly from C# in RAD, Unit testing etc. Scenarios
* Certain constructs are designed for direct from C# in these situations, e.g. FSharp.Core.FuncConvert.

It is important that the library and its implementation be an exemplar for F# coding and design guidelines.

### Design Goal: A Good .NET Citizen (Alignment and Sharing Considerations)

F# must be a good .NET citizen and consider ramifications w.r.t.

* 4.0 BigInt
* 4.0 Quotations <-> Expression trees & statement trees
* Tuple
* Set, Map (immutable collections)
* Async

In particular, the introduction of FSharp.Core.dll into .NET programming practice (even as its own DLL) raises questions about future possible BCL-led proposals for shared types or functionality. The full list of primary types defined in FSharp.Core.dll is as follows

* Tuple<...>
* Lazy<T>
* FSharp.Core.FSharpFunc<T,U> (first-class function values with efficient decurrying)
* FSharp.Core.FSharpRef<T> (single-cell mutable reference cells)
* FSharp.Core.FSharpOption<T> (functional options)
* FSharp.Core.FSharpList<T>  (functional lists)
* FSharp.Collections.FSharpSet<T>  (functional sets)
* FSharp.Collections.FSharpMap<T> (functional dictionaries)
* FSharp.Reflection.UnionCase (reflective view of cases of an F# discriminated union)
* FSharp.Reflection.FSharpValue (reflective view of F#-specific details of F# values)
* FSharp.Reflection.FSharpType  (reflective view of F#-specific details of F# types)
* FSharp.Control.Async<T>
* FSharp.Control.MailboxProcessor<T>

Each of these types has been annotated with a section dealing with alignment and sharing considerations.

This is covered in more detail later in this specification.

### Design Goal: One Way to do Things

Programming can suffer from a proliferation of techniques to "encode" a problem into a language.

To some extent F# suffers from this due to

* Its mixed functional/OO nature
* The presence of constructs such as sequence expressions that give general purpose syntax for encoding certain computations.
* Early-F# compat heritage, both in library and in community know-how

However a major design goal of this library is to eliminate and reduce this phenomenon. For example early F# library had both Int32.to\_float and Float.of\_int. In F# these (and many other functions) are replaced by one (parametrically overloaded) conversion function "float".

One particular reason for doing this is related to meta-programming. When implementing an embedded language it is important that "canonical" functions such as conversion functions exist to form the leaf constructs of the embedded language.

### Design Goal: Enable a F# -style Functional Programming Language Subset

The F# style of functional programming language has particular characteristics for its core library constructs. These are

* A focus on programming with tuples, options, lists, arrays, records and discriminated unions
* A heavy reliance on "second order aggregate operators", notably the "standard F# compositional functional operators" catalogued in the table later in this specification
* Idiomatic use of curried functions to enable compositional programming with these operators

### Design goal: Binary Size

It is a strong requirement that this DLL should not grow in size from its current size (900K)

It is a long-term goal to have this DLL be < 500K.

### Design goal: Compiled Names follow .NET Standards

The assembly uses to CompiledName attribute throughout to ensure the binary compiled names of functional programming operators follow .NET standards. This was a requirement of the BCL teams to leave open the long-term possibility to ship FSharp.Core.dll as part of the .NET framework.

## Non-design goals

This section documents the goals that do not need to be achieved with this DLL. Other DLLs such as the FSharp.PowerPack.dll (an open source component) fill this role.

### Non-design goal: Backwards Compatibility with Early F# and ML variations

It is not a design goal that the library achieves backwards-compatibility with “research” versions of F#.

### Non-design goal: FxCop Perfection

This library is not for use by other .NET languages. Thus it includes public constructs that do not meet .NET library design standards.

Thus perfection w.r.t. FxCop rules is not achieved. For example, exemptions are applied for the following:

* CA1709, Library naming for unqualified names (e.g. abs, sin, cos) and library naming for F# functional operators (e.g. List.map)
* Type variable naming. F# uses short type variable names, and after a lengthy discussion by the language design team, the decision was that this is and remains appropriate for F#.
* Various FxCop rules trigger as false positives with F# code, e.g.
  + Unused code. Generated property members are unused for private types, but the F# compiler must generate these in case F# reflection is used over the type.
* The following naming clashes:
  + Option is flagged as a keyword. However the name is standard F# practice and, as a library construct, replaces the corresponding feature in other languages. It will not change.
  + Event is flagged as a keyword. However its role is as a library construct, and replaces the corresponding feature in other languages.
  + Set is flagged as a keyword (in C#). However the word “Set” has a clear and expected mathematical meaning that corresponds to its use in F#

See the separate document “FxCop and F#” for full details.

### Non-design goal: Minimalism

It is possible to imagine a version of FSharp.Core.dll that contains very few constructs, e.g. just, say, tuple types, function types, list type and option type, with the aim of being as lightweight/minimal as possible.

However this is not a design goal of this library.

## Historical API Design Work

As we’ve been in the process of bringing F# toward product quality, we’ve done a lot of API design work.  Here are some of the major items we’ve pursued, for reference:

Reduced library surface area

We removed 60% of constructs from the F# library, including big rationals, several functional programming types, matrix and vector types and numerous compatibility functions. These are now shipped as an open source compatibility component called the FSharp.PowerPack

Sharing with .NET

We moved to use the .NET 4.0 Tuple, Lazy and BigInteger types

Naming

F# uses two naming styles: standard, implementation-oriented functional programming uses single word lower case, with ocaml\_case for some compound words. However all OO APIs are written using PascalCase and follow .NET naming conventions, for example the F# reflection and quotation APIs are OO APIs using this convention.

.NET Design Consistency

We’ve made an effort to align areas such as FSharp.Reflection and FSharp.Quoations with their related .NET APIs, System.Reflection and System.Linq.Expressions.

Full FXCop Review

This has resulted in many minor API improvements.

# Design Guidelines

## Standard F# Operators

The functional programming subset implemented by F# is heavily based around the use of "standard F# functional operators". These effectively have the status of keywords in the F# language, though are reimplemented afresh for different types using a functional programming module.

The full list of standard operators used in this library is shown below.

Query Operators

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.choose f c | ('T -> option<'U>) -> C<'T> -> C<'U> |
| C.collect f c | ('T -> C<'U>) -> C<'T> -> C<'U> |
| C.exists f c | ('T -> bool) -> C<'T> -> bool |
| C.exists2 f c1 c2 | ('T -> 'U -> bool) -> C<'T> -> C<'U> -> bool |
| C.find f c | ('T -> bool) -> C<'T> -> 'T |
| C.filter f c | ('T -> bool) -> C<'T> -> C<'T> |
| C.tryFind f c | ('T -> bool) -> C<'T> -> int |
| C.forall f c | ('T -> bool) -> C<'T> -> bool |
| C.forall2 f c1 c2 | ('T -> 'U -> bool) -> C<'T> -> C<'U> -> bool |
| C.map f c | ('T -> 'U) -> C<'T> -> C<'U> |
| C.map2 f c1 c2 | ('T1 -> 'T2 -> 'U) -> C<'T1> -> C<'T2> -> C<'U> |
| C.mapi f c | (int -> 'T -> 'U) -> C<'T> -> C<'U> |
| C.mapi2 f c1 c2 | (int -> 'T1 -> 'T2 -> 'U) -> C<'T1> -> C<'T2> -> C<'U> |
| C.partition f c | ('T -> bool) -> C<'T> -> C<'T> \* C<'T> |
| C.pick f c | ('T -> option<'U>) -> C<'T> -> 'U |
| C.tryFind f c | ('T -> bool) -> C<'T> -> 'T option |
| C.tryFindIndex f c | ('T -> bool) -> C<'T> -> int option |
| C.tryPick f c | ('T -> option<'U>) -> C<'T> -> option<'U> |

Zip, Unzip Operators

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.unzip c | C<'T1 \* 'T2> -> C<'T1> \* C<'T2> |
| C.unzip3 c | C<'T1 \* 'T2 \* 'T3> -> C<'T1> \* C<'T2> \* C<'T3> |
| C.zip c | C<'T1> \* C<'T2> -> C<'T1 \* 'T2> |
| C.zip3 c | C<'T1> \* C<'T2> \* C<'T3> -> C<'T1 \* 'T2 \* 'T3> |

Folding, Scanning, Reducing Operators

|  |  |
| --- | --- |
|  |  |
| C.fold f state c | ('State -> 'T -> 'State) -> 'State -> C<'T> -> 'State |
| C.fold2 f state c1 c2 | ('State -> 'T1 -> 'T2 -> 'State) -> 'State -> C<'T1> -> C<'T2> -> 'State |
| C.foldBack f state c | ('T -> 'State -> 'State) -> C<'T> -> 'State -> 'State |
| C.foldBack2 f state c1 c2 | ('T1 -> 'T2 -> 'State -> 'State) -> C<'T1> -> C<'T2> -> 'State -> 'State |
| C.reduce f c | ('T -> 'T -> 'T) -> C<'T> -> 'T |
| C.reduceBack f c | ('T -> 'T -> 'T) -> C<'T> -> 'T |
| C.scan f c | ('State -> 'T -> 'T) -> 'State -> C<'T> -> 'T |
| C.scanBack f c | ('T -> 'State -> 'T) -> C<'T> -> 'State -> 'T |

Iteration Operators

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.iter f c | ('T -> unit) -> C<'T> -> unit |
| C.iter2 f c1 c2 | ('T -> 'U -> unit) -> C<'T> -> C<'U> -> unit |
| C.iteri f c | (int -> 'T -> unit) -> C<'T> -> unit |
| C.iteri2 f c1 c2 | (int -> 'T -> 'U -> unit) -> C<'T> -> C<'U> -> unit |

Aggregation Operators

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.average c | C<'T> -> 'T  when 'T : (static ( + ) : 'T \* 'T -> 'T)  and 'T : (static DivideByInt : 'T \* int -> 'T)  and 'T : (static Zero : 'T) |
| C.averageBy c | ('T -> 'U') -> C<'T> -> 'U  when 'U : (static ( + ) : 'U \* 'U -> 'U)  and 'U : (static DivideByInt : 'U \* int -> 'U)  and 'U : (static Zero : 'U) |
| C.sum c | C<'T> -> 'T  when 'T : (static ( + ) : 'T \* 'T -> 'T)  and 'T : (static Zero : 'T) |
| C.sumBy c | ('T -> 'U') -> C<'T> -> 'U  when 'U : (static ( + ) : 'U \* 'U -> 'U)  and 'U : (static Zero : 'U) |

Sorting Operators

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.sort c | C<'T> -> C<'T> |
| C.sortBy c | ('T -> 'Key) -> C<'T> -> C<'T> |
| C.sortWith c | ('T -> 'T -> int) -> C<'T> -> C<'T> |

Grouping Operators

See the Seq module

Operators for Linearly Ordered and Indexed Types (strings, lists, arrays etc.)

|  |  |
| --- | --- |
| Operator Name | Typical signature |
| C.append c1 c2 | C -> C -> C |
| C.concat cs | seq<C> -> C |
| C.empty | C |
| C.init f n | (int -> 'T) -> int -> C<'T> |

## Use of Tuples

Within the functional programming subset, tuples are consistently used in preference to fresh nominal types.

Note that tuples introduce a "design rigidity" since new members can't be added. However for the canonical functional programming subset no future variance is expected in these types.

## Use of F# Function Values

Since the API is for use from F# programs, the use of F# functions (FSharpFunc<\_,\_>) is encouraged throughout instead of nominal delegate types.

Note: F# functions differ from System.Func<\_,\_> because the latter doesn't support efficient curried invocation, an essential feature for general purpose functional programming, and critically important for the idioms of F#.

If non-trivial technical design changes were made to CLI implementations then it would, in theory possible to unify FSharpFunc<\_,\_> and Func<\_,\_>. A Dev10 request was made for this by F# but not auctioned by the CLR team. See the section on "Alignment considerations" in the FSharpFunc<\_,\_> section for brief details on this.

Notably

* F# functions "T -> T -> int" are used for comparison
* F# functions "T -> int" are used for hashing

## Currying Guidelines

The use of currying is restricted to

* the functional programming modules
* the F# top level operators

It follows the guidelines in Expert F# Chapter 19.

## Overloading Guidelines and Type Inference

APIs designed for use from F# use overloading far less than other .NET APIs. This is because

* Overloading can interact badly with type inference
* Overloading is needed less through the use of F# optional parameters
* Functional languages tend to prefer "parametric" overloading, present in F# through member constraints

Overloading may not be used with the functional programming operators since they are defined in modules.

There are enormous tradeoffs between type inference and overloading as two ways to achieve succinctness in programming language design. The C# and general .NET approach is, of course, to use overloading ubiquitously at the API boundary.

The F# approach is very different: a core of the language and library uses a rigid, non-overloaded approach to language design. This extends beyond the functional operator core. As a result

* overloading is never used in F# functional operator design
* overloading is used rarely in F# OO API design
* where it is used it tends to be based on argument count

For example, consider this:

let mapSquare f xs = List.map (List.map f) xs

Contrast against the following definition where we assume a typical overloaded "OO" member called Map:

let mapSquare (f: 'T -> 'U) (xs: list<list<'T>>) = xs.Map(fun (x:list<'T>) -> x.Map(f))

If the "Map" function is overloaded (e.g. in the style of LINQ's Select) then one or more type annotations like those shown are likely to be needed to resolve the overloading. This can be extremely destructive to F# programming practice in the core of the language.

Indeed, a number of F# language features such as optional arguments and member constraints are very much designed to reduce the need for overloading in API design.

As a result, .NET member overloading is used rarely in F# API design, and where it is used careful attention must be given to the type inference ramifications.

## Capitalization Guidelines

**No lower case names may be used except in the functional programming operators. In all other cases .NET/F#-OO guidelines must be followed.**

This DLL adopts F# capitalization guidelines for F#-facing DLLs. In summary:

* PascalCase:
  + all library type names
  + all members
* lower case:
  + F# top level operators
  + the "standard F# compositional functional programming operators" in the core functional programming modules: String, Set, Map, List, Seq, Option, Event

See below for discussion of ruby\_case v. camelCase.

## Underscore Guidelines

**No underscores may be used except in the functional programming operators. In all other cases .NET/F#-OO guidelines must be followed.**

Underscores have traditionally been very heavily used in early F# programming. The style is extremely pervasive in some quarters of F# programming and is likely to constitute a significant part of F# development methodology for a substantial number of highly vocal and influential users and community members. For example, Jon Harrop's book F# for Scientists used underscores throughout, though his later work now uses camelCase.

Since 2007 there has been a long standing effort by the F# team to reduce and minimize the use of underscores throughout the F# library. This has resulted in the total elimination of underscores from the library itself.

## Abbreviation Guidelines

F# functional operator design eschews overloading. Instead abbreviations are used. This is widely popular and is an integral part of functional programming in a type inferred language.

Thus the following uses of abbreviations are tolerated in the design of the functional operators in FShar.Core.dll

<many abbreviations for top level operators, e.g. sin>

List.rev reverse

List.mapi map-indexed

List.iter iterate

List.iteri iterate-indexed

List.init initialize

List.nth integer index into list (n'th)

Note: Some of these abbreviations have been the subject of much discussion in the F# team. Ultimately we decided we were willing to recognize the heritage of F# in cases such as this and just leave things as they are.

Note: the use of abbreviations goes against .NET style guidelines. This tension is a difficult point to resolve, though ultimately our choice has been to lean towards succinctness and away from longer names and/or overloading.

# FSharp.Core – Basic Operators

## Basic Type Abbreviations

|  |  |
| --- | --- |
| Type Name | Short Description |
| obj | System.Object |
| exn | System.Exception |
| nativeint | System.IntPtr |
| unativeint | System.UIntPtr |
| string | System.String |
| float32, single | System.Single |
| float, double | System.Double |
| sbyte | System.SByte |
| byte | System.Byte |
| int16 | System.Int16 |
| uint16 | System.UInt16 |
| int32, int | System.Int32 |
| uint32 | System.UInt32 |
| int64 | System.Int64 |
| uint64 | System.UInt64 |
| char | System.Char |
| bool | System.Boolean |
| decimal | System.Decimal |

## Basic Arithmetic Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| (+) | **x + y** | Overloaded addition |
| (-) | **x - y** | Overloaded subtraction |
| (\*) | **x \* y** | Overloaded multiplication |
| (/) | **x / y** | Overloaded division |
| (%) | **x % y** | Overloaded modulus |
| (~-) | **-x** | Checked overloaded unary negation |
| not | **not x** | Boolean negation |

## Generic Equality and Comparison Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| (<) | **x < y** | Generic less-than |
| (<=) | **x <= y** | Generic less-than-or-equal |
| (>) | **x > y** | Generic greater-than |
| (>=) | **x >= y** | Generic greater-than-or-equal |
| (=) | **x = y** | Generic equality |
| (<>) | **x <> y** | Generic disequality |
| max | **max x y** | Generic maximum |
| min | **min x y** | Generic minimum |

## Bitwise manipulation operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| (<<<) | **x <<< y** | Overloaded bitwise shift-left |
| (>>>) | **x >>> y** | Overloaded bitwise arithmetic shift-right |
| (^^^) | **x ^^^ y** | Overloaded bitwise exclusive or |
| (&&&) | **x &&& y** | Overloaded bitwise and |
| (|||) | **x ||| y** | Overloaded bitwise or |
| (~~~) | **~~~x** | Overloaded bitwise negation |

## Math operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| abs | **abs x** | Overloaded absolute value |
| acos | **acos x** | Overloaded inverse cosine |
| asin | **asin x** | Overloaded inverse sine |
| atan | **atan x** | Overloaded inverse tangent |
| atan2 | **atan2 x y** | Overloaded inverse tangent of x/y |
| ceil | **ceil x** | Overloaded floating point ceiling |
| cos | **cos x** | Overloaded cosine |
| cosh | **cosh x** | Overloaded hyperbolic cosine |
| exp | **exp x** | Overloaded exponent |
| floor | **floor x** | Overloaded floating point floor |
| log | **log x** | Overloaded natural logarithm |
| log10 | **log10 x** | Overloaded base-10 logarithm |
| (\*\*) | **x \*\* y** | Overloaded exponential |
| pown | **pown x y** | Overloaded integer exponential |
| round | **round x** | Overloaded rounding |
| sign | **sign x** | Overloaded sign function |
| sin | **sin x** | Overloaded sine function |
| sinh | **sinh x** | Overloaded hyperbolic sine function |
| sqrt | **sqrt x** | Overloaded square root function |
| tan | **tan x** | Overloaded tangent function |
| tanh | **tanh x** | Overloaded hyperbolic tangent function |

## Function Pipelining and Composition Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| (|>) | **x |> f** | Pipelining |
| (||>) | **(x,y) ||> f** | Two-input pipelining |
| (|||>) | **(x,y,z) |||> f** | Three-input pipelining |
| (>>) | **f >> g** | Function composition |
| (<|) | **f <| x** | Backward pipelining |
| (<<) | **g << f** | Backward function composition |
| ignore | **ignore x** | Compute and discard a value |

## Object Transformation Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| box | **box x** | Convert to object representation |
| hash | **hash x** | Generic hashing operator |
| sizeof | **sizeof<*type*>** | Compute the size of a value of the given type |
| typeof | **typeof<*type*>** | Compute the **Type** representation of the given type |
| typedefof | **typedefof<*type*>** | Compute the **Type** representation of the given type and calls GetGenericTypeDefinition if this is a generic type. |
| unbox | **unbox x** | Convert form object representation |
| ref | **ref x** | Allocate a mutable reference cell |
| (!) | **!x** | Read a mutable reference cell |

## Pair Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| fst | **fst p** | Take the first element of a pair |
| snd | **snd p** | Take the second element of a pair |

## Exception Operators

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| failwith | **failwith x** | Raise a **FailureException** exception |
| invalidArg | **invalidArg arg msg** | Raise an **ArgumentException** exception |
| invalidOp | **invalidOp msg** | Raise an **InvalidOperationException** exception |
| raise | **raise x** | Raise an exception |
| rethrow | **rethrow()** | Special operator to raise an exception |

## Input/Output Handles

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| stdin | **stdin** | Computes System.Console.In |
| stdout | **stdout** | Computes System.Console.Out |
| stderr | **stderr** | Computes System.Console.Error |

## Overloaded Conversion Functions

The following operators are defined in **Microsoft.FSharp.Core.Operators**:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| byte | **byte x** | Overloaded converstion to a byte |
| sbyte | **sbyte x** | Overloaded converstion to a signed byte |
| int16 | **int16 x** | Overloaded converstion to a 16 bit integer |
| uint16 | **uint16 x** | Overloaded converstion to an unsigned 16 bit integer |
| int32, int | **int32 x**  **int x** | Overloaded converstion to a 32 bit integer |
| uint32 | **uint32 x** | Overloaded converstion to an unsigned 32 bit integer |
| int64 | **int64 x** | Overloaded converstion to a 64 bit integer |
| uint64 | **uint64 x** | Overloaded converstion to an unsigned 64 bit integer |
| nativeint | **nativeint x** | Overloaded converstion to an native integer |
| unativeint | **unativeint x** | Overloaded converstion to an unsigned native integer |
| float, double | **float x**  **double x** | Overloaded converstion to a 64-bit IEEE floating point number |
| float32, single | **float32 x**  **single x** | Overloaded converstion to a 32-bit IEEE floating point number |
| decimal | **decimal x** | Overloaded converstion to a System.Decimal number |
| char | **char x** | Overloaded converstion to a System.Char value |
| enum | **enum x** | Overloaded converstion to a typed enumeration value |

## Checked Arithmetic Operators

The module **Microsoft.FSharp.Core.Operators.Checked** defines runtime-overflow-checked versions of the following operators:

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| (+) | **x + y** | Checked overloaded addition |
| (-) | **x – y** | Checked overloaded subtraction |
| (\*) | **x \* y** | Checked overloaded multiplication |
| (~-) | **-x** | Checked overloaded unary negation |
| byte | **byte x** | Checked overloaded converstion to a byte |
| sbyte | **sbyte x** | Checked overloaded converstion to a signed byte |
| int16 | **int16 x** | Checked overloaded converstion to a 16 bit integer |
| uint16 | **uint16 x** | Checked overloaded converstion to an unsigned 16 bit integer |
| int32, int | **int32 x**  **int x** | Checked overloaded converstion to a 32 bit integer |
| uint32 | **uint32 x** | Checked overloaded converstion to an unsigned 32 bit integer |
| int64 | **int64 x** | Checked overloaded converstion to a 64 bit integer |
| uint64 | **uint64 x** | Checked overloaded converstion to an unsigned 64 bit integer |
| nativeint | **nativeint x** | Checked overloaded converstion to an native integer |
| unativeint | **unativeint x** | Checked overloaded converstion to an unsigned native integer |
| char | **char x** | Checked overloaded converstion to a System.Char value |

## Type Abbreviations

See the F# Language specification for the full collection of operators.

|  |  |  |
| --- | --- | --- |
| Type Name | Example | Short Description |
| list<'T> | **int list** | FSharp.Core.List<'T> |
| seq<'T> | **seq<int>** | System.Collections.Generic.IEnumerable<'T> |
| ResizeArray<'T> | **ResizeArray<int>** | System.Collections.Generic.List<'T> |

## Special Types

### byref<\_>

This is a fake type, compiled to byref annotations in parameter passing positions. See the F# language specification

### float<\_>, float32<\_>, decimal<\_>

These are fake types, corresponding to System.Double, System.Single and System.Decimal with units-of-measure annotations

### nativeptr<\_>

This is a fake type, erased by the F# compiler. See the F# language specification



## FSharp.Core.MatchFailure (Exception)

Non-exhaustive match failures will raise Match failures

exception MatchFailure of [string](Microsoft.FSharp.Core.type_string.html) \* [int](Microsoft.FSharp.Core.type_int.html) \* [int](Microsoft.FSharp.Core.type_int.html)

## Attribute and Attribute Flag Types

See the F# Language Specification

# FSharp.Core - Basic Types

## FSharp.Core.Ref<T> (Type, compiled name FSharpRef<T>)

This is a type of mutable cells of type T allocated on the heap

Design Criteria

The F# design criteria are:

* Reference semantics
* It can be nested, e.g. Ref<Ref<int>>
* A a single public mutable instance field called “contents”
* A requirement that the underlying field must be public so we can take the address of it when using reference cells a boxes for mutable arguments passed as byref parameters and used a "l-values" in mutation operators

Performance Criteria

N/A

Naming

F# has some naming constraints for this type:

* It is lower-case “ref” for F# programmers (we use a type abbreviation to get this effect)
* The upper case name is not generally used by F# programmers and could potentially be changed. ReferenceCell<T> would be one option.

Alignment Considerations

This has superficial similarity to other types I’ve seen around such as DLR’s StrongBox<T>

Usage Model

**ref 4**

**r := 5**

**!r**

**r.Value**

**r.Value <- 5**

**r.contents**

**r.contents <- 5**

Signature

type Ref<'T> = {mutable contents: 'T;}

member Value: 'T with get, set

## FSharp.Core.Option<T> (Type+Module)

The type of optional values. This type implements the F# discriminated union "Some of 'T | None".

Design Criteria

* it can be nested, e.g. Option<Option<int>>
* it is immutable
* it is an F# discriminated union. This makes it technically difficult to share: F# really treats this type as special
* it specifically uses “null” to represent None, a fact carefully baked into the F# compiler and essential for efficiency of many functional programming algorithms

Naming

F# has some naming constraints for this type:

* It is lower-case “option” for F# programmers (we use a type abbreviation to get this effect)
* The upper case name is not generally used by F# programmers and could potentially be changed.

Performance Criteria

The main performance issue is that this is a reference type.

Representation

'None' values will appear as the value 'null' to other .NET languages. Instance methods (including getters for instance properties) on this type will appear as static methods to other .NET languages due to the use of 'null' as a value representation.

Alignment Considerations

The type has superficial similarity to Nullable<T>. However that that type may not be nested due to an unfortunate design limitation added to .NET generics.

Usage Model

**None**

**Some v**

**opt.Value**

**opt.IsSome**

**opt.IsNone**

**opt |> Option.collect f**

**opt |> Option.exists f**

**opt |> Option.filter f**

**opt |> Option.fold f state**

**state |> Option.foldBack f opt**

**(state,opt) ||> Option.fold f**

**(opt,state) ||> Option.foldBack f**

**opt |> Option.forall f**

**opt |> Option.get**

**opt |> Option.isNone**

**opt |> Option.isSome**

**opt |> Option.iter f**

**opt |> Option.map f**

**opt |> Option.toArray**

**opt |> Option.toList**

Signature

[<CompilationRepresentation(CompilationRepresentationFlags.UseNullAsTrueValue)>]

type Option<'T> =

| None

| Some of 'T

member Value: 'T

member IsSome: [bool](Microsoft.FSharp.Core.type_bool.html)

member IsNone: [bool](Microsoft.FSharp.Core.type_bool.html)

static None: 'T [option](Microsoft.FSharp.Core.type_option.html)

static Some: 'T -> 'T [option](Microsoft.FSharp.Core.type_option.html)

Signature (module)

module Option =

val collect: ('T -> 'U option) -> 'T option -> 'U option

val exists: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> 'T option -> bool

val fold: ('U -> 'T -> 'U) -> 'U -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> 'U

val foldBack: ('T -> 'U -> 'U) -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> 'U -> 'U

val forall: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> [bool](Microsoft.FSharp.Core.type_bool.html)

val get: 'T option -> 'T

val isNone: 'T option -> [bool](Microsoft.FSharp.Core.type_bool.html)

val isSome: 'T option -> [bool](Microsoft.FSharp.Core.type_bool.html)

val iter: ('T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> 'T option -> [unit](Microsoft.FSharp.Core.type_unit.html)

val map: ('T -> 'U) -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> 'U option

val toArray: 'T [option](Microsoft.FSharp.Core.type_option.html) -> array<'T>

val toList: 'T [option](Microsoft.FSharp.Core.type_option.html) -> list<'T>

~~val filter: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> 'T [option](Microsoft.FSharp.Core.type_option.html)~~

~~val length: 'T [option](Microsoft.FSharp.Core.type_option.html) -> [int](Microsoft.FSharp.Core.type_int.html)~~

~~val partition: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> 'T [option](Microsoft.FSharp.Core.type_option.html) -> 'T [option](Microsoft.FSharp.Core.type_option.html) \* 'T [option](Microsoft.FSharp.Core.type_option.html)~~

## FSharp.Core.String (Module)

A minimal set of functional programming operators related to processing and transforming strings.

Design Criteria

* Implement standard F# operators that are commonly useful on strings
* However, strings also implement IEnumerable and can thus be used as sequences of characters. Thus not all of the functional programming operators need to be supplied specifically on string.

Performance Criteria

TBD

Usage Model

"abc"

@"c:\foo.txt"

"""abc"def"hij""" (proposed, perhaps for F# V2)

s1 + s2

+ .NET usage

str.Length

String.init 10 (fun i -> char i)

String.create 10 ' '

strs |> String.concat ","

str |> String.exists f

str |> String.forall f

str |> String.iter f

str |> String.iteri f

str |> String.length

str |> String.map f

str |> String.mapi f

str |> String.collect f

str |> String.split [' '; '\t'; '\n'; '\r']

Signature (Module)

module String =

val collect: ([char](Microsoft.FSharp.Core.type_char.html) -> [string](Microsoft.FSharp.Core.type_string.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [string](Microsoft.FSharp.Core.type_string.html)

val concat: string -> seq<string> -> string

val exists: ([char](Microsoft.FSharp.Core.type_char.html) -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> bool

val forall: ([char](Microsoft.FSharp.Core.type_char.html) -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [bool](Microsoft.FSharp.Core.type_bool.html)

val iter: ([char](Microsoft.FSharp.Core.type_char.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri: (int -> [char](Microsoft.FSharp.Core.type_char.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val init: (int -> string) -> int -> [string](Microsoft.FSharp.Core.type_string.html)

val length: [string](Microsoft.FSharp.Core.type_string.html) -> [int](Microsoft.FSharp.Core.type_int.html)

val map: ([char](Microsoft.FSharp.Core.type_char.html) -> [char](Microsoft.FSharp.Core.type_char.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [string](Microsoft.FSharp.Core.type_string.html)

val mapi: (int -> [char](Microsoft.FSharp.Core.type_char.html) -> [char](Microsoft.FSharp.Core.type_char.html)) -> [string](Microsoft.FSharp.Core.type_string.html) -> [string](Microsoft.FSharp.Core.type_string.html)

val replicate: int -> char -> string

# FSharp.Collections (Namespace)

## Design Considerations

We expect these collections to be learnt in the following orders:

* List
* List+Array
* List+Array+Seq
* List+Array+Seq+Set+Map
* List+Array+Seq+ResizeArray

The following functionality relations exist:

* Array <= List <= Seq

People using sets and maps are expected to know about “Seq”. For example, to build a new set from an old set we expect either “s |> Seq.choose ... |> Set.ofSeq” or “set (seq { for x in s -> (x,x) })” etc.

## FSharp.Collections.List (Type+Module)

Design Criteria

This is a functional type implementing leftist-linked lists, primarily for use in F# implementation code.

* It is an F# discriminated union
* It does not use “null” to represent empty (to allow the type to implement IEnumerable<\_>)

Naming

F# has some naming constraints for this type:

* Naming constraints:
  + It is lower-case “list” for F# programmers (we use a type abbreviation to get this effect)
  + The upper case name is not generally used by F# programmers and could potentially be changed.

Performance Criteria

This type is not the highest performance collection (in that case arrays should be used instead)

Serialization

TBD

Alignment

This type doesn’t really seem a candidate for sharing. The incremental value to imperative .NET programmers is low given the availability of other collections. Indeed F# would prefer to see an immutable array type added to the .NET libraries.

Usage Model

[]

3 :: xs

[1;2;3]

[ for x in 1..3 -> x + 1 ]

[ for x in 1..3 do

for y in 4..6 do

if x + y % 3 = 0 then

yield (x,y) ]

xs.Head

xs.Tail

xs.IsEmpty

xs.Length

xs.IsCons

List.append

List.average

List.averageBy

List.choose

List.collect

List.concat

List.empty

List.exists

List.exists2

List.filter

List.find

List.fold

List.fold2

List.foldBack

List.foldBack2

List.forall

List.forall2

List.head

List.init

List.isEmpty

List.iter

List.iter2

List.iteri

List.iteri2

List.length

List.map

List.map2

List.map3

List.mapi

List.mapi2

List.max

List.maxBy

List.min

List.minBy

List.nth

List.ofArray

List.ofSeq

List.partition

List.reduce

List.reduceBack

List.rev

List.scan

List.scanBack

List.sort

List.sortBy

List.sortWith

List.tail

List.toArray

List.toSeq

List.tryPick

List.tryFind

List.unzip

List.unzip3

List.zip

List.zip3

Signature (Type)

type List<'T> =

| ( [] )

| ( :: ) of 'T \* List<list<'T>>

interface IEnumerable<'T>

member Head: 'T

member Tail: 'T [list](Microsoft.FSharp.Collections.type_list.html)

member IsEmpty: [bool](Microsoft.FSharp.Core.type_bool.html)

member IsCons: bool

member Length: int

static Empty: list<'T>

static Cons: 'T \* list<'T> -> list<'T>

type list<'T> = List<'T>

Signature (Module)

module List =

val append: list<'T> -> list<'T> -> list<'T>

val average: Numeric<'T> => list<'T> -> 'T

val averageBy: Numeric<'U> => ('T -> 'U) -> list<'T> -> 'U

val collect: ('T -> list<'U>) -> list<'T> -> list<'U>

val concat: seq<list<'T>> -> list<'T>

val empty: list<'T>

val exists: ('T -> bool) -> list<'T> -> bool

val exists2: ('T1 -> 'T2 -> bool) -> list<'T1> -> list<'T2> -> bool

val filter: ('T -> bool) -> list<'T> -> list<'T>

val find: ('T -> bool) -> list<'T> -> 'T

val fold: ('State -> 'T -> 'State) -> 'State -> list<'T> -> 'State

val fold2: ('State -> 'T1 -> 'T2 -> 'State) -> 'State -> list<'T1> -> list<'T2> -> 'State

val foldBack: ('T -> 'State -> 'State) -> list<'T> -> 'State -> 'State

val foldBack2: ('T1 -> 'T2 -> 'State -> 'State) -> list<'T1> -> list<'T2> -> 'State -> 'State

val forall: ('T -> bool) -> list<'T> -> bool

val forall2: ('T1 -> 'T2 -> bool) -> list<'T1> -> list<'T2> -> bool

val head: list<'T> -> 'T

val init: int -> (int -> 'T) -> list<'T>

val isEmpty: list<'T> -> bool

val iter: ('T -> unit) -> list<'T> -> unit

val iter2: ('T1 -> 'T2 -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> 'T1 list -> 'T2 list -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> list<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri2: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T1 -> 'T2 -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> 'T1 list -> 'T2 list -> [unit](Microsoft.FSharp.Core.type_unit.html)

val length: list<'T> -> [int](Microsoft.FSharp.Core.type_int.html)

val map: ('T -> 'U) -> list<'T> -> list<'U>

val map2: ('T1 -> 'T2 -> 'U) -> 'T1 list -> 'T2 list -> list<'U>

val map3: ('T1 -> 'T2 -> 'T3 -> 'U) -> 'T1 list -> 'T2 list -> 'T3 list -> list<'U>

val mapi: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> 'U) -> list<'T> -> list<'U>

val mapi2: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T1 -> 'T2 -> 'U) -> 'T1 list -> 'T2 list -> list<'U>

val max: list<'T> -> 'T

val maxBy: ('T -> 'U) -> list<'T> -> 'T

val min: list<'T> -> 'T

val minBy: ('T -> 'U) -> list<'T> -> 'T

val nth: list<'T> -> int -> 'T

val ofArray: array<'T> -> list<'T>

val ofSeq: seq<'T> -> list<'T>

val partition: ('T -> bool) -> list<'T> -> list<'T> \* list<'T>

val reduce: ('T -> 'T -> 'T) -> list<'T> -> 'T

val reduceBack: ('T -> 'T -> 'T) -> list<'T> -> 'T

val rev: list<'T> -> list<'T>

val scan: ('State -> 'T -> 'State) -> 'State -> list<'T> -> list<'State>

val scanBack: ('T -> 'State -> 'State) -> list<'T> -> 'State -> list<'State>

val sort: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [list](Microsoft.FSharp.Collections.type_seq.html)<'T>

val sortBy: ('T -> 'Key) -> [list](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [list](Microsoft.FSharp.Collections.type_seq.html)<'T>

val sortWith: ('T -> 'T -> int) -> list<'T> -> list<'T>

val sum: Numeric<'T> => list<'T> -> 'T

val sumBy: Numeric<'U> => ('T -> 'U) -> list<'T> -> 'U

val tail: list<'T> -> list<'T>

val toArray: list<'T> -> array<'T>

val toSeq: list<'T> -> seq<'T>

val tryPick: ('T -> 'U option) -> list<'T> -> 'U option

val tryFind: ('T -> bool) -> list<'T> -> 'T option

val trymap: ('T -> 'U option) -> list<'T> -> list<'U>

val unzip: ('T1 \* 'T2) list -> list<'T1> \* list<'T2>

val unzip3: ('T1 \* 'T2 \* 'T3) list -> list<'T1> \* list<'T2> \* 'T3 list

val zip: list<'T1> -> list<'T2> -> ('T1 \* 'T2) list

val zip3: list<'T1> -> list<'T2> -> 'T3 list -> ('T1 \* 'T2 \* 'T3) list

~~val tryFind\_index: ('T -> bool) -> list<'T> -> int option~~

~~val tryFind\_indexi: (int -> 'T -> bool) -> list<'T> -> int option~~

~~val find\_index: ('T -> bool) -> list<'T> -> int~~

~~val find\_indexi: (int -> 'T -> bool) -> list<'T> -> int~~

## FSharp.Collections.Array (Module)

Design Criteria

* Support standard F# functional operators for the .NET array type, as if it were immutable
* Support additional operators

Performance Criteria

TBD

Serialization

The serialization formats of this type are dicated by the .NET implementation(s) of serialization.

Naming

F# has some naming constraints for this type:

* Naming constraints:
  + It is lower-case “int array” for F# programmers, or "int[]"
  + The upper case name is not generally used by F# programmers.

Alignment

The .NET System.Array module has an adhoc and cumbersome set of operators, some of them related to functional programming. Some (such as binary search) are suitable for direct use from F#, but many others are cumbersome, unsafe and overly imperative.

Usage Model

1-dimensional (generic) arrays.

module Array =

val append: array<'T> -> array<'T> -> array<'T>

val average: array<'T> -> 'T

val averageBy: ('T -> 'U) -> array<'T> -> 'U

val blit: array<'T> -> int -> array<'T> -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val choose: ('T -> 'U option) -> array<'T> -> 'U

val concat: [seq](Microsoft.FSharp.Collections.type_seq.html)<array<'T>> -> array<'T>

val collect: ('T -> array<'U>) -> array<'T> -> array<'U>

val copy: array<'T> -> array<'T>

val create: [int](Microsoft.FSharp.Core.type_int.html) -> 'T -> array<'T>

val empty: array<'T>

val exists: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> bool

val exists2: ('T -> 'U -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> array<'U> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val fill: array<'T> -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)

val filter: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> array<'T>

val find: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> 'T

val tryPick: ('T -> 'U option) -> array<'T> -> 'U option

val reduce: ('T -> 'T -> 'T) -> array<'T> -> 'T

val reduceBack: ('T -> 'T -> 'T) -> array<'T> -> 'T

val fold: ('T -> 'U -> 'T) -> 'T -> array<'U> -> 'T

val fold2: <signature omitted>

val foldBack: ('T -> 'U -> 'U) -> array<'T> -> 'U -> 'U

val foldBack2: <signature omitted>

val forall: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val forall2: <signature omitted>

val init: [int](Microsoft.FSharp.Core.type_int.html) -> ([int](Microsoft.FSharp.Core.type_int.html) -> 'T) -> array<'T>

val isEmpty: 'T [array](Microsoft.FSharp.Core.type_array.html) -> bool

val iter: ('T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> array<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iter2: ('T -> 'U -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> array<'T> -> array<'U> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> array<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri2: <signature omitted>

val length: array<'T> -> [int](Microsoft.FSharp.Core.type_int.html)

val map: ('T -> 'U) -> array<'T> -> array<'U>

val map2: ('T -> 'U -> 'V) -> array<'T> -> array<'U> -> 'V array

val mapi: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> 'U) -> array<'T> -> array<'U>

val mapi2: <signature omitted>

val max: array<'T> -> 'T

val maxBy: ('T -> 'U) -> array<'T> -> 'T

val min: 'T [list](Microsoft.FSharp.Collections.type_list.html) -> 'T

val minBy: ('T -> 'U) -> array<'T> -> 'T

val ofList: list<'T> -> array<'T>

val ofSeq: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> array<'T>

val partition: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> array<'T> \* array<'T>

val permute: [Permutation](Microsoft.FSharp.Collections.type_Permutation.html) -> array<'T> -> array<'T>

val rev: array<'T> -> array<'T>

val scan: ('U -> 'T -> 'U) -> 'U -> array<'T> -> array<'U>

val scanBack: ('T -> 'U -> 'U) -> array<'T> -> 'U -> array<'U>

val sort: ('T -> 'T -> [int](Microsoft.FSharp.Core.type_int.html)) -> array<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val sub: array<'T> -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> array<'T>

val sum: Numeric<'T> => array<'T> -> 'T

val sumBy: Numeric<'U> => ('T -> 'U) -> array<'T> -> 'U

val toList: array<'T> -> list<'T2>

val toSeq: array<'T> -> seq<'T>

val tryFind: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> array<'T> -> 'T option

val trymap: ('T -> 'U option) -> array<'T> -> array<'U>

val unzip: array<'T \* 'U> -> array<'T> \* array<'U>

val zip: array<'T> -> array<'U> -> array<'T \* 'U>

val zeroCreate: [int](Microsoft.FSharp.Core.type_int.html) -> array<'T>

## FSharp.Collections.Array.Parallel (Module)

Design Criteria

TBD

Performance Criteria

* All operators can accept arrays that occupy 1/4 the physical memory of a machine
* Array.Parallel.map achieves successful parallelization (compared to Array.map) for a "update-a-reference-cell-100-times" function on very large imput arrays (assuming no compiler optimization removes the work from Array.map ☺)

#time

let r = ref 3

let work \_ =

for i = 0 to 100 do

r := !r + 1

let arr = Array.init<int> 30000000 id

#time

arr |> Array.iter work

System.Threading.Parallel.ForEach(arr,(fun \_ -> work()))

System.Threading.Parallel.For(0,30000000,(fun \_ -> work()))

* Likewise everything except Array.Parallel.choose, Array.Parallel.filter and Array.Parallel.collect for comparable workloads.
* Array.Parallel.choose and Array.Parallel.collect
* Successful parallelization on 2-10 element input arrays and a long running computation

Naming

Also considered FSharp.Parallel.Array.\*

Alignment

Exposes 4.0 mscorlib goodness in an F# way

Usage Model

arr |> Array.Parallel.choose f

arr |> Array.Parallel.collect f

arr |> Array.Parallel.map f

arr |> Array.Parallel.mapi f

arr |> Array.Parallel.iter f

arr |> Array.Parallel.iteri f

Array.Parallel.init n f

Signature

module Array =

module Parallel =

val choose : chooser:('T -> option<'U>) -> array:array<'T> -> array<'U>

val collect : ('T -> array<'U>) -> array<'T> -> array<'U>

val map : ('T -> 'U) -> array<'T> -> array<'U>

val mapi : (int -> 'T -> 'U) -> array<'T> -> array<'U>

val iter : ('T -> unit) -> array<'T> -> unit

val iteri : (int -> 'T -> unit) -> array<'T> -> unit

val init : int -> (int -> 'T) -> array<'T>

Sample Implementation (checked for performance criteria, not correctness)

open System.Threading

let map f (x : 'a array) =

let results = Array.zeroCreate x.Length

Parallel.For(0,x.Length,(fun i -> results.[i] <- f(x.[i])))

results

let iter f (arr : 'a array) =

Parallel.ForEach(arr,(fun elem -> f elem))

let mapi f (x : 'a array) =

let f2 = OptimizedClosures.FastFunc2<\_,\_,\_>.Adapt(f)

let results = Array.zeroCreate x.Length

Parallel.For(0,x.Length,(fun i -> results.[i] <- f2.Invoke(i,x.[i])))

results

let iteri f (x : 'a array) =

let f = OptimizedClosures.FastFunc2<\_,\_,\_>.Adapt(f)

//let results = Array.zeroCreate x.Length

Parallel.For(0,x.Length,(fun i -> f.Invoke(i,x.[i])))

let init n f =

let results = Array.zeroCreate n

Parallel.For(0,n,(fun i -> results.[i] <- f i))

results

let choose f (c: array<\_>) =

// Note: this array of bools should be a bitarray

let results1 = System.Collections.BitArray c.Length

let results2 = Array.zeroCreate<\_> c.Length

Parallel.For(0,c.Length,(fun i ->

match f c.[i] with

| None -> ()

| Some v ->

results1.[i] <- true;

results2.[i] <- v))

let mutable n = 0

for i = 0 to arr.Length - 1 do

if results1.[i] then

n <- n + 1

let results = Array.zeroCreate n

let mutable curr = 0

for i = 0 to arr.Length - 1 do

if results1.[i] then

results.[curr] <- results2.[i]

curr <- curr + 1

results

let filter f (c:array<\_>) =

let results1 = System.Collections.BitArray c.Length

let results2 = Array.zeroCreate<\_> c.Length

Parallel.For(0,c.Length,(fun i -> if f c.[i] then results1.[i] <- true))

let mutable n = 0

for i = 0 to arr.Length - 1 do

if results1.[i] then

n <- n + 1

let results = Array.zeroCreate n

let mutable curr = 0

for i = 0 to arr.Length - 1 do

if results1.[i] then

results.[curr] <- results2.[i]

curr <- curr + 1

results

## FSharp.Collections.Seq (Module)

Design Criteria

* Provide functional programming operators for the seq<\_> type (i.e. IEnumerable<\_>)

Performance Criteria

TBD

Usage Model

TBD

Serialization

The serialization formats of instances are determined by their underlying implementation

Sequences implemented by F# sequences shall be marked Serializable in the .NET metadata.

Signature

TBD

val append: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val average: Numeric<'T> => seq<'T> -> 'T

val averageBy: Numeric<'U> => ('T -> 'U) -> seq<'T> -> 'U

val cache: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> Seq.CachedSeq<'T>

val choose: ('T -> 'U [option](Microsoft.FSharp.Core.type_option.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>

val collect: ('T -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>

val collect2: ('T -> 'U -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'V>) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U> -> seq<'V>

val compare: ('T -> 'T -> [int](Microsoft.FSharp.Core.type_int.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [int](Microsoft.FSharp.Core.type_int.html)

val concat: [seq](Microsoft.FSharp.Collections.type_seq.html)<#seq<'V>> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val delay: ([unit](Microsoft.FSharp.Core.type_unit.html) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val empty: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val exists: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val exists2: ('T -> 'U -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val filter: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val find: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'T

val first: ('T -> 'U [option](Microsoft.FSharp.Core.type_option.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'U [option](Microsoft.FSharp.Core.type_option.html)

val fold: ('U -> 'T -> 'U) -> 'U -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'U

val fold2: ('U -> 'T1 -> 'T2 -> 'U) -> 'U -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T1> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T2> -> 'U

val forall: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val forall2: ('T -> 'U -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val head: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'T

val init: [int](Microsoft.FSharp.Core.type_int.html) -> ([int](Microsoft.FSharp.Core.type_int.html) -> 'T) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val initInfinite: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val iter: ('T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iter2: ('T -> 'U -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri2: <standard signature>

val length: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [int](Microsoft.FSharp.Core.type_int.html)

val map: ('T -> 'U) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>

val map2: ('T -> 'U -> 'V) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'V>

val mapi: ([int](Microsoft.FSharp.Core.type_int.html) -> 'T -> 'U) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>

val mapi2: <standard signature>

val reduce: ('T -> 'T -> 'T) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'T

val nth: [int](Microsoft.FSharp.Core.type_int.html) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> 'T

val ofArray: 'T [array](Microsoft.FSharp.Core.type_array.html) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val ofList: 'T [list](Microsoft.FSharp.Collections.type_list.html) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val sortBy: ('T -> 'Key) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val sort: ('T -> 'T -> bool) -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val pairwise: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T \* 'T>

val windowed: int -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<array<'T>>

val readonly: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val scan: ('U -> 'T -> 'U) -> 'U -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'U>

val singleton: 'T -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val sum: Numeric<'T> => 'T [list](Microsoft.FSharp.Collections.type_list.html) -> 'T

val sumBy: Numeric<'U> => ('T -> 'U) -> seq<'T>-> 'U

val max: seq<'T> -> 'T

val maxBy: ('T -> 'U) -> seq<'T> -> 'T

val min: seq<'T> -> 'T

val minBy: ('T -> 'U) -> seq<'T> -> 'T

val skip: int -> seq<'T> -> seq<'T>

val take: [int](Microsoft.FSharp.Core.type_int.html) -> seq<'T> -> seq<'T>

val skipWhile: ('T -> bool) -> seq<'T> -> seq<'T>

val takeWhile: ('T -> bool) -> seq<'T> -> seq<'T>

val toArray: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> array<'T>

val toList: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> list<'T>

val truncate: [int](Microsoft.FSharp.Core.type_int.html) -> seq<'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'T>

val tryFind: ('T -> bool) -> seq<'T> -> 'T option

val unfold: ('U -> ('T \* 'U) option) -> 'U -> seq<'T>

val cast: IEnumerable -> seq<'T>

val zip: seq<'T> -> seq<'U> -> seq<'T \* 'U>

val zip3: seq<'T> -> seq<'U> -> seq<'V> -> seq<'T \* 'U \* 'V>

## FSharp.Collections.Set (Type+Module)

Set<T> implements

* immutable sets
* using an underlying binary tree representation
* has good memory sharing characteristics on incremental update
* sealed type
* implements IEnumerable<\_>, ICollection<\_>

Comparison is F# generic comparison, potentially using implementations of the IComparable interface on key values.

Likewise Map<\_,\_> implements immutable dictionaries, using an underlying binary tree representation, with good incremental memory characteristics.

These sets can be used with elements of any type, but you should check that structural hashing and equality on the element type are correct for your type.

Alignment considerations

The Set and Map types are for implementing algorithms and communicating results between agents and threads. They do not appear in APIs.

Future-proofing

Both the Set<\_> and Map<\_,\_> types have potential utility for C# programmers. As a result they could be a candidate for a BCL-led initiative to add functional collections to mscorlib. F# would accept (indeed welcome with open arms!) a BCL implementation of the type. Requirements would be:

* in mscorlib
* permanently maintains immutability semantics
* provides the (fairly minimalist) OO functionality available on the current F# types
* has performance at least as good as the current implementation, including sharing internal nodes on incremental updates
* had an API reasonable to use from F# (though F# would maintain its existing Set.\* API)

Performance Criteria

TBD

Efficiency: Structural comparison is relatively efficient but is not a suitable choice in all circumstances, e.g. it may not compare efficiently on non-reference types and deeply-structured types.

Serialization

TBD.

Usage Model

**new Set<\_>(seq)**

**set.Contains key**

**set.Add (key)**

**set.Count**

**set.IsEmpty**

**set.Remove key**

**set1 = set2**

**set1 + set2**

**set1 - set2**

**set1 < set2**

**hash set**

**Set.empty**

**Set.singleton x**

**Set.union s1 s2**

**Set.intersect s1 s2**

**set |> Set.add key**

**set |> Set.exists f**

**set |> Set.filter f**

**set |> Set.forall f**

**set |> Set.count**

**set |> Set.isEmpty**

**set |> Set.fold f state**

**state |> Set.foldBack f set**

**(state,set) ||> Set.fold f**

**(set,state) ||> Set.foldBack f**

**sets |> Set.intersectAll**

**sets |> Set.unionAll**

**set |> Set.iter f**

**set |> Set.map f**

**set |> Set.partition f**

**set |> Set.remove x**

**set |> Set.toArray**

**set |> Set.toList**

**set |> Set.toSeq**

**arr |> Set.ofArray**

**xs |> Set.ofList**

**xs |> Set.ofSeq**

Signature (Type)

type Set<'T> =

interace IComparable

interface ICollection<'T>

member Add: 'T -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

member Contains: 'T -> [bool](Microsoft.FSharp.Core.type_bool.html)

member Count: [int](Microsoft.FSharp.Core.type_int.html)

member IsEmpty: [bool](Microsoft.FSharp.Core.type_bool.html)

member IsSubsetOf: Set<'T> -> bool

member IsSupersetOf: Set<'T> -> bool

member MinimumElement: 'T

member MaximumElement: 'T

member Remove: 'T -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

static Empty: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

static ( + ): [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> \* [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

static ( - ): [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> \* [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

Signature (Module)

[<CompilationRepresentation(CompilationRepresentationFlags.ModuleSuffix)>]

module FSharp.Collections.Set =

val add: 'T -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val count: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [int](Microsoft.FSharp.Core.type_int.html)

val empty: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val exists: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val filter: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val fold: ('U -> 'T -> 'U) -> 'U -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> 'U

val foldBack: ('T -> 'U -> 'U) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> 'U -> 'U

val forall: ('T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val intersectAll: [seq](Microsoft.FSharp.Collections.type_seq.html)<[Set](Microsoft.FSharp.Collections.type_Set.html)<'T>> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val intersect: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val unionAll: [seq](Microsoft.FSharp.Collections.type_seq.html)<[Set](Microsoft.FSharp.Collections.type_Set.html)<'T>> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val union: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val isEmpty: [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val iter: ('T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val map: ('T -> 'U) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'U>

val mem: 'T -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val ofArray: 'T [array](Microsoft.FSharp.Core.type_array.html) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val ofList: 'T [list](Microsoft.FSharp.Collections.type_list.html) -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

val ofSeq: seq<'T> -> Set<'T>

val partition: ('T -> bool) -> Set<'T> -> Set<'T> \* Set<'T>

val remove: 'T -> Set<'T> -> Set<'T>

val singleton: 'T -> Set<'T>

val subset: Set<'T> -> Set<'T> -> bool

val toArray: Set<'T> -> array<'T>

val toList: Set<'T> -> list<'T>

val toSeq: Set<'T> -> seq<'T>

## FSharp.Collections.Map<Key,Value> (Type)

Design Criteria

Map<Key,Value> implements

* immutable associative lookup tables
* uses an underlying binary tree representation
* has good memory sharing characteristics on incremental update
* is a sealed type
* implements IEnumerable<\_>, ICollection<\_>

Comparison is F# generic comparison, potentially using implementations of the IComparable interface on key values.

Naming

We considered a better name than “Map”. However any replacement name must be

* Short (c.f. Map.xyz)
* Better than Map
* Communicate the distinct functional nature of the collection

Suggestions were “Mapping”, “Table”, “Keyed”, "ReadonlyDictionary", "TreeMap", "Lookup". In the end we stuck with "Map".

Performance Criteria

Performance comparison points:

* Dictionary<\_,\_>
* Clojure hash maps

Maps based on structural comparison are only efficient for small keys (i.e. keys with fast comparison). They are not a suitable choice if keys are recursive data structures or require non-structural comparison semantics.

Serialization

TBD

Alignment and Future-proofing

See Set<\_>

Usage Model

**new Map<\_,\_>(seq)**

**tab.[key]**

**tab.Add (key,value)**

**tab.Count**

**tab.ContainsKey key**

**tab.IsEmpty**

**tab.Remove key**

**tab.TryFind key**

**tab1 = tab2**

**tab1 < tab2**

**hash tab**

**Map.empty**

**tab |> Map.add key value**

**tab |> Map.exists f**

**tab |> Map.filter f**

**tab |> Map.find key**

**tab |> Map.first f**

**tab |> Map.forall f**

**tab |> Map.isEmpty**

**tab |> Map.iter f**

**tab |> Map.map f**

**tab |> Map.mapi f**

**tab |> Map.mem x**

**tab |> Map.partition f**

**tab |> Map.remove x**

**tab |> Map.toArray**

**tab |> Map.toList**

**tab |> Map.toSeq**

**arr |> Map.ofArray**

**xs |> Map.ofList**

**xs |> Map.ofSeq**

**(state, tab) ||> Map.fold f**

**(tab, state) ||> Map.foldBack f**

Signature (type)

type Map<'Key,'T>

interface IDictionary<'Key,'T>

new: [seq](Microsoft.FSharp.Collections.type_seq.html)<'Key \* 'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

member (.[]): 'Key -> 'T with get

member IsEmpty: [bool](Microsoft.FSharp.Core.type_bool.html)

member Count: [int](Microsoft.FSharp.Core.type_int.html)

member TryFind: 'Key -> 'T [option](Microsoft.FSharp.Core.type_option.html)

member Remove: 'Key -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

member Find: 'Key -> 'T

static Empty: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

member ContainsKey: 'Key -> [bool](Microsoft.FSharp.Core.type_bool.html)

member Add: 'Key \* 'T -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

Signature (module)

module Map =

val add: 'Key -> 'T -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val empty: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val exists: ('Key -> 'T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val filter: ('Key -> 'T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val find: 'Key -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> 'T

val first: ('Key -> 'T -> 'U [option](Microsoft.FSharp.Core.type_option.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> 'U [option](Microsoft.FSharp.Core.type_option.html)

val fold: ('Key -> 'T -> 'V -> 'V) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> 'V -> 'V

val foldBack: ('Key -> 'T -> 'V -> 'V) -> Map<'Key,'T> -> 'V -> 'V

val forall: ('Key -> 'T -> [bool](Microsoft.FSharp.Core.type_bool.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val isEmpty: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val iter: ('Key -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [unit](Microsoft.FSharp.Core.type_unit.html)

val map: ('Key -> 'T -> 'U) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'U>

val mapi: ('Key -> 'T -> 'U) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'U>

val mem: 'Key -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [bool](Microsoft.FSharp.Core.type_bool.html)

val ofArray: ('Key \* 'T) [array](Microsoft.FSharp.Core.type_array.html) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val ofList: ('Key \* 'T) [list](Microsoft.FSharp.Collections.type_list.html) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val ofSeq: [seq](Microsoft.FSharp.Collections.type_seq.html)<'Key \* 'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val partition:('Key -> 'T->[bool](Microsoft.FSharp.Core.type_bool.html)) -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> \* [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T>

val remove: 'Key -> Map<'Key,'T> -> Map<'Key,'T>

val toArray: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> ('Key \* 'T) [array](Microsoft.FSharp.Core.type_array.html)

val toList: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> ('Key \* 'T) [list](Microsoft.FSharp.Collections.type_list.html)

val toSeq: [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> [seq](Microsoft.FSharp.Collections.type_seq.html)<'Key \* 'T>

val tryFind: 'Key -> [Map](Microsoft.FSharp.Collections.type_Map.html)<'Key,'T> -> 'T [option](Microsoft.FSharp.Core.type_option.html)

## FSharp.Collections.Array2D (Module)

A minimalist set of functions for strongly type operations on rank-2 array types

module Array2D =

val base1: 'T[,] -> [int](Microsoft.FSharp.Core.type_int.html)

val base2: 'T [[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> [int](Microsoft.FSharp.Core.type_int.html)

val blit: source:'T[[,]](Microsoft.FSharp.Core.type_array.html) -> sourceIndex1: int -> sourceIndex2:[int](Microsoft.FSharp.Core.type_int.html) -> target:'T [[,]](Microsoft.FSharp.Core.type_array.html) ->

targetIndex1:int -> targetIndex2:[int](Microsoft.FSharp.Core.type_int.html) -> length1:int -> length2:[int](Microsoft.FSharp.Core.type_int.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val copy: 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

val create: int -> int -> 'T -> 'T[,]

val init: int -> int -> (int -> int -> 'T) -> 'T[,]

val initBased: [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> ([int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

val iter: ('T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val iteri: ([int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

val length1: 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> [int](Microsoft.FSharp.Core.type_int.html)

val length2: 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> [int](Microsoft.FSharp.Core.type_int.html)

val map: ('T -> 'U) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> 'U[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

val mapi: ([int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T -> 'U) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> 'U[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

val rebase: 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

val zeroCreate: [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T[[,]](../fslib/Microsoft.FSharp.Core.type__%5b,%5d.html)

## FSharp.Collections.Array3D (Module)

A minimalist set of functions for strongly type operations on rank-3 array types

TBD

## FSharp.Collections.Array4D (Module)

A minimalist set of functions for strongly type operations on rank-4 array types

TBD

## Convenience Functions (FSharp.Core.ExtraTopLevelOperators Module, AutoOpen)

Additional bindings available at the top level

### Collection Functions

Performance Criteria

TBD

Signature

val dict: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T \* 'U> -> IDictionary<'T,'U>

val set: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> -> [Set](Microsoft.FSharp.Collections.type_Set.html)<'T>

### Printing Functions (FSharp.Core.ExtraTopLevelOperators Module)

The F# typesafe "printf" feature is heavily used in F# code and the following operators are defined by default for printing and formatting:

Summary

|  |  |  |
| --- | --- | --- |
| Operator/Function Name | Expression Form | Short Description |
| printf | printf "%s" msg | Format to stdout |
| printfn | printfn "%s" msg | Format to stdout, with newline |
| fprintf | fprintf tw "%s" msg | Format to TextWriter |
| fprintfn | fprintfn tw "%s" msg | Format to TextWriter, with newline |
| eprintf | eprintf "%s" msg | Format to standard error |
| eprintfn | eprintfn "%s" msg | Format to standard error, with newline |
| sprintf | sprintf "%s" msg | Format to string |
| failwithf | failwithf "%s" msg | Format to Failure exception |

Signature

[<AutoOpen>]

module Microsoft.FSharp.Core.Pervasives =

...

val eprintf: TextWriterFormat<'T> -> 'T

val eprintfn: TextWriterFormat<'T> -> 'T

val failwithf: TextWriterFormat<'T,'d> -> 'T

val fprintf : [TextWriter](http://msdn2.microsoft.com/en-us/library/System.IO.TextWriter.aspx) -> TextWriterFormat<'T> -> 'T

val fprintfn: [TextWriter](http://msdn2.microsoft.com/en-us/library/System.IO.TextWriter.aspx) -> TextWriterFormat<'T> -> 'T

val printf: TextWriterFormat<'T> -> 'T

val printfn: TextWriterFormat<'T> -> 'T

val sprintf: StringFormat<'T> -> 'T

# FSharp.Control – Events and Observables

**Design Background (Events)**

.NET events are revealed to F# code as instances of the type IEvent. The delegate type parameter and arguments are determined by the F# type checker based on information stored for the .NET event. F# code may also declare .NET events over any .NET delegate type. For example, IEvent<PaintEventHandler,PaintEventArgs> can be used for an event whose callbacks are of type PaintEventHandler. The delegate and argument types must match.

F# gives special status to non-virtual instance member properties compatible with type IDelegateEvent, generating approriate .NET metadata to make the member appear to other .NET languages as a .NET event.

Fresh .NET-compatible events declared in F# are typically values of type IEvent<'T>. These hold event handlers of type Handler<'T>.

**Design Background (Observables)**

.NET 4.0 observables are a model of "unbuffered event streams". Conceptually, imagine someone throwing balls at you very fast, even when you are not ready. You are allowed to spawn additional arms to catch the balls and do something with them. Observables are the basis of the Rx Reactive Framework.

Observables have benefits over events with regards to subscription management and garbage collection of event chains. These benefits are often only noticeable on larger applications.

Design Criteria (Observables)

The F# design goal is to provide smooth transitions from Event based code to Observable-based code.

Design Criteria

* Support F# first class event and observable values
* Provide standard F# operators where appropriate
* Allow event values to be published as standard .NET metadata

Dependency Basis

The API surface area of the model requires the following .NET types

* System.IObserver<'T> (with members OnNext, OnError, OnCompleted)
* System.IObservable<'T> (with members Subscribe)

Usage Model

event |> Event.map (fun evArgs -> ...)

event |> Event.filter (fun evArgs -> ...)

...

event.Add callback

event.AddHandler delegate

event.RemoveHandler delegate

event.Subscribe callback

observable |> Observable.map (fun evArgs -> ...)

observable |> Observable.filter (fun evArgs -> ...)

...

observable.Add callback

observable.Subscribe callback

Signature (Types)

type IDelegateEvent<'Del> =

abstract member RemoveHandler: 'Del -> unit

abstract member AddHandler: 'Del -> unit

type IEvent<'Del,'Args> =

inherit [IPrimitiveDelegateEvent](Microsoft.FSharp.Control.type_IPrimitiveDelegateEvent.html)<'Del>

inherit IObservable<'Args>

type IEvent<'Args> = IEvent<Handler<'T>, 'T>

type Handler<'T> = delegate of sender: [obj](Microsoft.FSharp.Core.type_obj.html) \* args: 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)

Signature (Module)

module Event =

val add: ('T -> unit) -> IObservable<'T> -> unit

val choose: ('T -> 'U option) -> IEvent<'Del,'T> -> IEvent<'U>

val filter: ('T -> bool) -> IEvent<'Del,'T> -> IEvent<'T>

val scan: ('U -> 'T -> 'U) -> 'U -> IEvent<'Del,'T> -> IEvent<'U>

val map: ('T -> 'U) -> IEvent<'Del,'T> -> IEvent<'U>

val merge: IEvent<'Del,'T> -> IEvent<'Del,'T> -> IEvent<'T>

val pairwise: IEvent<'Del,'T> -> IEvent<'T \* 'T>

val partition: ('T -> bool) -> IEvent<'Del,'T> -> IEvent<'T> \* IEvent<'T>

val split: ('T -> Choice<'U1,'U2>) -> IEvent<'Del,'T> -> IEvent<'U1> \* IEvent<'U2>

module Observable =

val add: ('T -> unit) -> IObservable<'T> -> unit

val choose: ('T -> 'U option) -> IObservable<'T> -> IObservable<'U>

val filter: ('T -> bool) -> IObservable<'T> -> IObservable<'T>

val scan: ('U -> 'T -> 'U) -> 'U -> IObservable<'T> -> IObservable<'U>

val map: ('T -> 'U) -> IObservable<'T> -> IObservable<'U>

val merge: IObservable<'T> -> IObservable<'T> -> IObservable<'T>

val pairwise: IObservable<'T> -> IObservable<'T \* 'T>

val partition: ('T -> bool) -> IObservable<'T> -> IObservable<'T> \* IObservable<'T>

val split: ('T -> Choice<'U1,'U2>) -> IObservable<'T> -> IObservable<'U1> \* IObservable<'U2>

val subscribe: ('T -> unit) -> IObservable<'T> -> IDisposable

# FSharp.Control – Asynchronous Programming

Design goals:

* Scalable, composable, non-blocking programming
* Task-based CPU and I/O parallelism

F# asynchronous workflows are a declarative, functional, compositional form of asynchronous programming.

Asynchronous computations are a computation, which, when run, will eventually produce a value of the given type, or else raise an exception. The value and/or exception is not returned to the caller immediately, but is rather passed to a continuation or exception continuation. Async computations can be specified using the F# 'workflow' syntax for building computations. Operationally, async computations typically run by scheduling micro operations in a queue-based scheduling engine (e.g. the .NET Thread Pool or the Parallel Extensions task processing queue). When waiting for asynchronous I/O, they are suspended as thunks, typically using ThreadPool.RegisterWaitForSingleObject, waiting for the I/O completion.

Some primitive asynchronous computations necessarily end up executing blocking operations: these should be run on threads specifically dedicated to resolving blocking conditions, for example via SwitchToNewThread. For example, FileOpen on Windows is, by design, a blocking operation. However frequently it is important to code as if this is asynchronous.

When run, async computations belong to an AsyncGroup. This can usually be specified when the async computation is started. The only action on an AsyncGroup is to raise a cancellation condition for the AsyncGroup. Async values check the cancellation condition for their AsyncGroup regularly, though synchronous computations within an asynchronous computation will not automatically check this condition

Implementation-wise, Async<T> involves no scheduling engine and rely on an underlying ClR scheduling engine such as the .NET thread pool or .NET futures. Using TPL futures is an area under active investigation.

Alignment considerations

The F# asynchronous workflow feature involves asynchronous programming and thus shares similarity with

* APM
* .NET 4.0 tasks
* CCR
* Reactive Framework IObservable
* Axum

Implementation-wise, Async<T> involves no scheduling engine and relies on the .NET thread pool.

Semantically, Task<T> is the most interesting comparison point.

* The type is semantically different to Task<T>. The following analogy is usually very helpful to digest:

*Async<T> is to Task<T> as IEnumerable<T> is to IEnumerator<T>*

That is, Task<T> can be seen as the execution substrate that underlies Async<T> programming. When you run an Async<T> you may be generating Task<T> values “under the hood”, just as when you iterate an IEnumerable<T> you may be generating one or more iterators under the hood. This analogy can be used to answer many design questions. For example, Async<T>.Current makes no sense.

Future-proofing

* F# asynchronous workflows continue to be influential with architects inside the company. Async workflow programming is being designed as a C#/VB language design extension.

## Dependency Basis

The API design depends on the following .NET 4.0 types. If these are not present, the F# library includes an implementation of the types

* System.AggregateException
* System.OperationCanceledException
* System.Threading. CancellationTokenRegistration
* System.Threading. CancellationToken
* System.Threading. CancellationTokenSource

On .NET 4.0, additional functionality is included related to .NET 4.0 tasks. This increases the dependency basics to include:

* System.Threading.Tasks.Task<T>

## Execution Semantics

Although the Async<T> type is trivial, its semantics are not.

### Threaded Context

When executing, the implicit state of an asynchronous computation contains

* the CancellationToken governing an execution of async
* the success continuation
* the exception continuation
* the cancellation continuation

### Cancellation

Asyncs support Unified Cancellation Model. All primitives starting execution of the async support specifying a System.Threading.CancellationToken that will govern execution of the async. Async primitives react to cancellation signal of governing cancellation token at frequent places, most notable on Binds.

When cancellation signal is raised and detected, asyncs cease execution and execute cancellation continuation of current threading context. Users can modify the cancellation continuation via Async.TryCanceled primitive.

The cancellation token governing the execution of an async should not be disposed until that execution ceases.

Cancellation token is available to excuting async via Async.CancellationToken primitive.

### Timeouts

There are functions in async library that start execution of an async and provide means for the caller to wait until execution is complete, for example Async.RunSynchronously (which blocks the caller until execution completes) and Async.StartChild (which returns an async that can be run to wait until the exexcution completes)

These functions typicallt allow specifiying timeouts for async execution. All those timeouts have a “soft” semantics: when timeout occurs, the cancellation signal for the executing async is raised, but the wait is not complete untile the computation reacts to the signal and ceases execution.

Example: System.Net.WebClient allows only one download request at a time.

Consider the following code:

let a = async {

      let! s = webClient.AsyncDownloadString(“<http://foobar.com>”)

      …

}

Async.RunSynchronously(a, 100)

Under the cooperative timeout discipline, if execution timeouts, it is guaranteed that webClient does not execute download request and is free to start another.

### Return to Synchronization Context

All primitives and compositions except Async.SwitchTo\* run their resulting continuation w.r.t. the same synchronization context the operation was begun in, as given by capturing System.Threading.SynchronizationContext.Current when the operation begins. If this is null, the continuation is run on a .NET thread pool thread.

Note: this refers to the SynchronizationContext.Current active at the *start* of execution of an Async<T>, rather than at the *creation* of the Async<T> object. Sometimes, these are the same, when an Async<T> is created and immediately started.

The following operations implicitly introduce a switch to the .NET Thread Pool for some or all child operations:

* Async.Start
* Async.StartChild
* Async.StartChildAsTask (depending on TaskCreationOptions)
* Async.StartAsTask (depending on TaskCreationOptions)
* Async.Parallel (for the component child tasks)
* Async.RunSynchronously (for the child task)

### Serialization and Cross-Machine/Cross-AppDomain programming:

Values of type Async<T> should be binary-serializable, as should the continuation objects provided by Async.FromContinuations(), as long as the closure of captured variables is binary-serializable.

This is particularly relevant to asynchronous, cross-app-domain programming using .NET remoting.

For example, this should be valid:

let SwitchToAppDomain (ad: System.AppDomain) : Async<unit> =

Async.FromContinuations(fun (cont,\_,\_) -> ad.DoCallBack(fun \_ -> cont()))

let report() = printfn "app domain '%s'" System.AppDomain.CurrentDomain.FriendlyName

let ad = System.AppDomain.CreateDomain "other"

async { report();

do! SwitchToAppDomain ad;

report() }

|> Async.RunSynchronously

Likewise this:

open Microsoft.FSharp.Control

let InOtherAppDomain (ad: System.AppDomain) (a: Async<'T>) : Async<'T> =

let result = AsyncResultCell<'T>()

async { ad.DoCallBack(fun \_ -> Async.StartWithContinuations(a,

continuation=(fun v -> result.RegisterResult (AsyncOk v)),

exceptionContinuation=(fun v -> result.RegisterResult (AsyncException v) ),

cancellationContinuation=(fun v -> result.RegisterResult (AsyncCanceled v) ) ) )

return! result.AsyncResult }

let report() = printfn "app domain '%s'" System.AppDomain.CurrentDomain.FriendlyName

let task = async { do report() }

let ad = System.AppDomain.CreateDomain "other"

task |> Async.RunSynchronously

task |> InOtherAppDomain ad |> Async.RunSynchronously

## FSharp.Control.Async<T>

The Async<T> type has no members and supports no interfaces. It is created by operations in the static Async type and through the workflow syntax enabled by the AsyncBuilder type.

Full Signature

type Async<'T>

## FSharp.Control. AsyncBuilder

The AsyncBuilder enables workflow syntax for Async<T>.

Performance Criteria

TBD

Usage Model

**async { return 3 }**

**async { ...**

**if ... then ...**

**... }**

**async { ...**

**if ... then ... else ...**

**... }**

**async { ...**

**match ... with**

**...**

**... }**

**async { ...**

**let x = ...**

**... }**

**async { ...**

**let! x = ...**

**... }**

**async { ...**

**for x in ... do**

**... }**

**async { ...**

**while ... do**

**... }**

**async { ...**

**try ...**

**finally ... }**

**async { ...**

**try ...**

**with ... }**

**async { use x = ... }**

**async { use! x = ... }**

Full Signature

type AsyncBuilder with

member Zero: [unit](Microsoft.FSharp.Core.type_unit.html) -> [Async](Microsoft.FSharp.Control.type_Async.html)<[unit](Microsoft.FSharp.Core.type_unit.html)>

member While: ([unit](Microsoft.FSharp.Core.type_unit.html) -> [bool](Microsoft.FSharp.Core.type_bool.html)) \* [Async](Microsoft.FSharp.Control.type_Async.html)<[unit](Microsoft.FSharp.Core.type_unit.html)> -> [Async](Microsoft.FSharp.Control.type_Async.html)<[unit](Microsoft.FSharp.Core.type_unit.html)>

member Using: 'T \* ('T -> Async<'U>) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'U> when 'T :> [IDisposable](http://msdn2.microsoft.com/en-us/library/System.IDisposable.aspx)

member TryFinally: Async<'T> \* ([unit](Microsoft.FSharp.Core.type_unit.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'T>

member Return: 'T -> Async<'T>

member For: [seq](Microsoft.FSharp.Collections.type_seq.html)<'T> \* ('T -> Async<[unit](Microsoft.FSharp.Core.type_unit.html)>) -> [Async](Microsoft.FSharp.Control.type_Async.html)<[unit](Microsoft.FSharp.Core.type_unit.html)>

member Delay: ([unit](Microsoft.FSharp.Core.type_unit.html) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'T>) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'T>

member Combine: Async<[unit](Microsoft.FSharp.Core.type_unit.html)> \* Async<'T> -> Async<'T>

member Bind: [Async](Microsoft.FSharp.Control.type_Async.html)<'T> \* ('T -> [Async](Microsoft.FSharp.Control.type_Async.html)<'U>) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'U>

## FSharp.Control. Async

The Async static class provides members which give additional rich functionality for async programming.

Performance Criteria

TBD

Usage Model

**async { ...**

**let! cancellationToken = AsyncCancellationToken**

**cancellationToken.Register (fun e -> printfn "I'm being cancelled")**

**... }**

**async { ...**

**let! subTask = Async.StartChild (async { ... })**

**... }**

**async { ...**

**do! Async.SwitchToThreadPool()**

**... }**

**async { ...**

**do! Async.SwitchToNewThread()**

**... }**

**async { ...**

**do! Async.SwitchToContext (syncContext)**

**... }**

Async.AsBeginEnd(computation)

Async.AwaitEvent(event, ?cancelAction)

Async.AwaitIAsyncResult(iar, ?millisecondsTimeout)

Async.AwaitTask(task)

Async.AwaitWaitHandle(waitHandle, ?millisecondsTimeout)

Async.CancellationToken: Async<CancellationToken>

Async.CancelDefaultToken() : unit

Async.Catch computation

Async.DefaultCancellationToken: CancellationToken

Async.FromBeginEnd(beginAction, endAction, ?cancelAction)

**Async.FromContinuations (fun (cont,econt,ccont) -> ...)**

Async.OnCancel()

Async.Parallel(computations)

Async.RunSynchronously(computation, ?timeout)

Async.Sleep(millisecondsTimeout)

Async.Start(computation, ?cancellationToken)

Async.StartAsTask(computation,?taskCreationOptions,?cancellationToken)

Async.StartChild(computation,millisecondsTimeout)

Async.StartChildAsTask(computation,?taskCreationOptions)

Async.StartImmediate(computation, ?cancellationToken)

Async.StartWithContinuations(computation,continuation,exceptionContinuation,

cancelContinuation,?cancellationToken)

Async.SwitchToContext(syncContext)

Async.SwitchToNewThread()

Async.SwitchToThreadPool()

Async.TryCancelled(computation,compensation)

### Async.AsBeginEnd

Signature

**TBD**

Descrption

TBD

Threading Guarantees

TBD

Examples

TBD

Async.AwaitWaitHandle(...)

### Async.AwaitEvent

Signature

**TBD**

Descrption

Return an asynchronous computation that waits for a single invocation of a .NET event by adding a handler to the event. Once the computation completes or is cancelled, the handler is removed from the event.

The computation will respond to cancellation while waiting for the event. If a cancellation occurs, and 'cancelAction' is specified, then it is executed, and the computation continues to wait for the event. If 'cancelAction' is not specified, then cancellation causes the computation to stop immediately, and if the event is subsequently raised it may be ignored.

Threading Guarantees

TBD

Examples

Waiting for a GUI event:

Async.AwaitEvent(form.Click)

### Async.AwaitIAsyncResult

Signature

**TBD**

Descrption

TBD

Threading Guarantees

TBD

Examples

TBD

Async.AwaitIAsyncResult(...)

### Async.AwaitTask

Signature

**TBD**

Descrption

Return an asynchronous computation that waits for the completion of the given task

Threading Guarantees

TBD

Examples

TBD

Async.AwaitTask(...)

### Async.AwaitWaitHandle

Signature

**TBD**

Descrption

Return an asynchronous computation that waits for the given WaitHandle to be set.

Threading Guarantees

TBD

Examples

TBD

Async.AwaitWaitHandle(...)

### Async.CancellationToken

Signature

**TBD**

### Async.CancelDefaultToken

Signature

**TBD**

### Async.Catch

Signature

**TBD**

Descrption

Return an asynchronous computation that waits for the given WaitHandle to be set.

Threading Guarantees

TBD

Examples

TBD

Async.Catch c

### Async.DefaultCancellationToken

Signature

**TBD**

### Async.FromBeginEnd

Signature

**static member FromBeginEnd(beginAction, endAction, ?cancelAction)**

**beginAction:(System.AsyncCallback \* obj -> System.IAsyncResult)**

**endAction:(System.IAsyncResult -> 'T)**

**?cancelAction : (unit -> unit)**

**Return: Async<'T>**

Descrption

Return an asynchronous computation in terms of a Begin/End pair of actions in the style used in .NET APIs where the overall operation is not qualified by any arguments.

When the computation is run, 'beginFunc' is executed, with a callback which represents the continuation of the computation. When the callback is invoked, the overall result is fetched using 'endFunc'.

The computation will respond to cancellation while waiting for the completion of the operation. If a cancellation occurs, and 'cancelAction' is specified, then it is executed, and the computation continues to wait for the completion of the operation. If 'cancelAction' is not specified, then cancellation causes the computation to stop immediately, and subsequent invocations of the callback are ignored.

Threading Guarantees

TBD

Examples

Using a web service:

Async.FromBeginEnd(ws.BeginGetWeather,ws.EndGetWeather)

Using an existing APM method that supports cancellation:

async {

try

// We specify req.Abort as the cancelAction. If successful, ir will cause

// a WebExceptionStatus.RequestCanceled to be raised from the web request.

return! Async.FromBeginEnd(beginAction=req.BeginGetResponse,

endAction = req.EndGetResponse,

cancelAction = req.Abort)

with

:? WebException as webExn

when webExn.Status = WebExceptionStatus.RequestCanceled ->

return! Async.FromContinuations (fun (\_,\_,ccont) ->

ccont (OperationCanceledException webExn.Message))

}

### Async.FromContinuations

Signature

**TBD**

### Async.OnCancel

Signature

**TBD**

### Async.Parallel

Signature

**TBD**

### Async.RunSynchronously

Signature

**TBD**

### Async.Sleep

Signature

**TBD**

### Async.Start

Signature

**TBD**

### Async.StartChild

Signature

**static member StartChild : computation:Async<'T> \* ?millisecondsTimeout : int -> Async<Async<'T>>**

Descrption

Start a child computation within an asynchronous workflow. This allows multiple asynchronous computations to be executed simultaneously.

This method should normally be used as the immediate right-hand-side of a 'let!' binding in an F# asynchronous workflow, i.e.,

async { ...

let! completor1 = childComputation1 |> Async.StartChild

let! completor2 = childComputation2 |> Async.StartChild

...

let! result1 = completor1

let! result2 = completor2

... }

When used in this way, each use of StartChild starts an instance of childComputation and returns a completor object representing a computation to wait for the completion of the operation. When executed, the completor awaits the completion of childComputation.

Exceptions

TBD

Edge Conditions

TBD

### Async.StartChildAsTask

Signature

**TBD**

### Async.StartImmediate

Signature

**TBD**

### Async.StartWithContinuations

Signature

**TBD**

Descrption

TBD

Threading Guarantees

TBD

Examples

TBD

Async.StartWithContinuations(...)

### Async.SwitchToNewThread

Signature

**TBD**

### Async.SwitchToThreadPool

Signature

**TBD**

### Async.TryCancelled

Signature

**TBD**

## FSharp.Control.CommonExtensions (Module, AutoOpen)

The extension methods in FSharp.Control.CommonExtensions provide F# asynchronous workflows corresponding for some of the APM patterns in mscorlib.dll.

Design Criteria

Asynchronous programming is much improved if the Begin/End APM pattern is routinely lifted through the use of an extension method.

Performance Criteria

TBD

Usage Model

**async { ...**

**let! bytes = stream.AsyncRead(numBytes)**

**... }**

**async { ...**

**do! stream.AsyncWrite(bytes)**

**... }**

**etc.**

Signature

TBD

type System.IO.Stream with

member AsyncRead: buffer:byte[] \* ?offset:int \* ?count:int -> Async<int>

member AsyncRead: int -> Async<byte[]>

member AsyncWrite: buffer:byte[] \* ?offset:int \* ?count:int -> Async<unit>

type System.Threading.WaitHandle with

member AsyncWaitOne: ?millisecondsTimeout:int -> Async<bool>

type System.Threading.Thread with

static AsyncSleep: millisecondsDueTime:int -> Async<unit>

## FSharp.Control.WebExtensions (Module, AutoOpen)

The extension methods in FSharp.Control.WebExtensions provide F# asynchronous workflows corresponding to web programming operations in mscorlib.dll.

Design Criteria

TBD

Performance Criteria

TBD

Usage Model

**async { ...**

**let! bytes = webRequest.AsyncGetResponse()**

**... }**

**async { ...**

**let! html = webClient.AsyncDownloadString()**

**... }**

## FSharp.Control.MailboxProcessor (Type)

This and its associated types is used to implement F# async actor agents using F# async workflow programming.

A MailboxProcessor is an asynchronous computation that includes the ability to read from a single dedicated channel (i.e. a single dedicated message queue). Anyone can send messages to a MailboxProcessor by using the Post method. A MailboxProcessor enters a state where it waits for the next message by calling its own Receive or TryReceive method. A MailboxProcessor can scan through all available messages using its own Scan or TryScan method, by using a function that selects an asynchronous computation to run based on a scan of the message queue. A MailboxProcessor is usually declared as a recursive object because it generally needs to use one or more of its own Receive, TryReceive, Scan or TryScan methods. It also often has to publish the ability for other asynchronous computations to send messages back to the MailboxProcessor.

Mailbox processing agents are always in-process.

Design Criteria

This type is carefully crafted for use from F# to give a good syntactic simulation of agent programming.

* It meets the F# design goal to support an approximation to Erlang agent programming.
* It goes hand in hand with F# async programming and is very much the rationale for the whole incorporation of async workflow programming into F#

Performance Criteria: Memory overhead

Running instances of the mailbox processing agents should have a minimum overhead of around 600 bytes or less per agent. Maximum cost of an agent is dictated by the cost of the closure objects and operating system handles that form its implementation.

Performance Criteria: Operating system handles

A mailbox processing agent that performs no other agent actions besides waiting on unlimited-timeout Receive should not require an operating system handle. Agents that perform Receives with timeouts may allocate an operating system handle.

Performance Criteria: Message processing rates

At the time of writing (Nov 2008) the implementation of mailbox processing agents consumes around 1million messages in 5 seconds on good hardware based on the following example code:

let counter =

new MailboxProcessor<\_>(fun inbox ->

let rec loop(n) =

async { if n > 1000000 then

printfn "n = %d, waiting..." n

let! msg = inbox.Receive()

return! loop(n+msg) }

loop(0))

counter.Start()

#time

for i in 0..1000000 do

counter.Post(1)

The message processing rate is not expected to reach the rates of systems dedicated entirely to actor-based processing.

Performance Criteria: Scheduling

Scheduling of agents is performed by a combination of the .NET thread pool and .NET futures. As such the F# team have little control over scheduling characteristics.

Design Criteria: Serialization

TBD

Alignment considerations

This type should be used to implement agents, and should not be used in APIs (not even F# APIs). As a result adding the type to some core library doesn’t seem critical, and alignment considerations are not paramount. If C# ever supports asynchronous, agent-based programming it would be realistic to implement a similar type crafted for use from C#.

Examples

The following example implements an integer register and is in some ways the "hello world" of actor processing:

let counter =

new MailboxProcessor<\_>(fun inbox ->

let rec loop(n) =

async { do printfn "n = %d, waiting..." n

let! msg = inbox.Receive()

return! loop(n+msg) }

loop(0))

// ----------------------------

counter.Start()

counter.Post(1)

counter.Post(2)

for i in 0..100 do

counter.Post(1)

Usage Model

**new MailboxProcessor<MessageType>(fun inbox ->**

**let rec state1(args) =**

**async { ... }**

**and state2(args) =**

**async { ... }**

**state1(initialArgs) )**

**agent.Start()**

**MailboxProcessor.Start(fun inbox ->**

**let rec state1(args) =**

**async { ... }**

**and state2(args) =**

**async { ... }**

**state1(initialArgs) )**

**agent.Post(message)**

**agent.PostAndReply(fun replyChannel -> message)**

**agent.PostAndAsyncReply(fun replyChannel -> message)**

**agent.PostAndTryAsyncReply(fun replyChannel -> message)**

**agent.CurrentQueueuLength**

**inbox.Receive()**

**inbox.Receive(timeout=N)**

**inbox.TryReceive()**

**inbox.TryReceive(timeout=N)**

**inbox.Receive()**

**inbox.Receive(timeout=N)**

**inbox.Scan ()**

**inbox.TryScan(timeout=N)**

**replyChannel.Reply(reply)**

Signature

type MailboxProcessor =

member DefaultTimeout: int with get,set

member Post: 'msg -> unit

member PostAndReply: (AsyncReplyChannel<'reply> -> 'msg) \* ?timeout:int -> 'reply

member PostAndAsyncReply: (AsyncReplyChannel<'reply> -> 'msg) \* ?timeout:int -> Async<'reply>

member PostAndTryAsyncReply: (AsyncReplyChannel<'reply> -> 'msg) \* ?timeout:int -> Async<'reply option>

member Receive: ?timeout:int -> Async<'msg>

member Scan: ('msg -> [Async](Microsoft.FSharp.Control.type_Async.html)<'res> [option](Microsoft.FSharp.Core.type_option.html)) \* ?timeout:[int](Microsoft.FSharp.Core.type_int.html) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'res>

member Start: [unit](Microsoft.FSharp.Core.type_unit.html) -> [unit](Microsoft.FSharp.Core.type_unit.html)

member TryReceive: ?timeout:[int](Microsoft.FSharp.Core.type_int.html) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'msg [option](Microsoft.FSharp.Core.type_option.html)>

member TryScan: ('msg -> [Async](Microsoft.FSharp.Control.type_Async.html)<'res> [option](Microsoft.FSharp.Core.type_option.html)) \* ?timeout:[int](Microsoft.FSharp.Core.type_int.html) -> [Async](Microsoft.FSharp.Control.type_Async.html)<'res option>

static Start:

(MailboxProcessor<'msg> -> [Async](Microsoft.FSharp.Control.type_Async.html)<[unit](Microsoft.FSharp.Core.type_unit.html)>) \* ?asyncGroup:[AsyncGroup](Microsoft.FSharp.Control.type_AsyncGroup.html) -> MailboxProcessor<'msg>

type AsyncReplyChannel<'reply> =

member Reply : value:'reply -> unit

# FSharp.Control – Lazy Programming

## FSharp.Control.Lazy (Type)

Design Criteria

This is a type of lazily-initialized (delayed) computations.

This type is now equated with .NET 4.0 System.Lazy<'T>

Performance Criteria

TBD

Usage Model

TBD

Additional Signature Surface Area Through Extension Members

TBD

type Lazy<'T> with

static member Create: ([unit](Microsoft.FSharp.Core.type_unit.html) -> 'T) -> [Lazy](Microsoft.FSharp.Control.type_Lazy.html)<'T>

static member CreateFromValue: 'T -> [Lazy](Microsoft.FSharp.Control.type_Lazy.html)<'T>

member Force: [unit](Microsoft.FSharp.Core.type_unit.html) -> 'T

type 'T lazy = [Lazy](Microsoft.FSharp.Control.type_Lazy.html)<'T>

# FSharp.Reflection (Namespace)

Design Criteria

* Enable reflection over F# union types, F# records types, F# tuple types and values of the same
* In the long term, provide a place to expose additional F# metadata through a core API

Usage Model

TBD

Alignment Considerations

We have assumed that there will be no BCL-led initiative for dealing with compiled encodings of discriminated union or record types. However even if there was the F# Reflection API could easily be updated to use this new compiled encoding.

## FSharp.Reflection.FSharpValue (Static Class)

This static class enables reflection over F# union values, F# record values and F# tuple values

Design Criteria

* Simple entry points to experiment with fetching and creating union, record and tuple values
* An entry point to allow the construction of a function value from a dynamic implementation
* Precomputation entry points to maximize the amount of computational effort that can be amortized priori to an inner loop that must construct or read many instances of F# union values, record values or tuple according to a fixed schema
* Allow analysis of private record and union values according to a passed-in bindingFlags parameter. No granularity greater than Public/NonPublic is required.

Usage Model

TBD

Performance Criteria

TBD

Signature

type FSharpValue =

static GetUnionCase: value:[obj](Microsoft.FSharp.Core.type_obj.html) -> [UnionCaseInfo](Microsoft.FSharp.Core.type_obj.html)

static GetRecordField: record:[obj](Microsoft.FSharp.Core.type_obj.html) \* info:PropertyInfo -> [obj](Microsoft.FSharp.Core.type_obj.html)

static GetRecordFields: record:[obj](Microsoft.FSharp.Core.type_obj.html) \* ?bindingFlags:BindingFlags -> [obj](Microsoft.FSharp.Core.type_obj.html) []

static GetTupleField: tuple:obj \* index:int -> [obj](Microsoft.FSharp.Core.type_obj.html)

static GetTupleFields tuple:[obj](Microsoft.FSharp.Core.type_obj.html) -> [obj](Microsoft.FSharp.Core.type_obj.html) []

static MakeFunction : functionType:Type \* implementation:(obj -> obj) -> obj

static MakeRecord : recordType:Type \* values:obj[] -> [obj](Microsoft.FSharp.Core.type_obj.html)

static MakeTuple : tupleElements: obj[] \* tupleType : Type -> obj

static MakeUnion : unionCase:UnionCaseInfo \* obj[] \* ?bindingFlags:BindingFlags -> [obj](Microsoft.FSharp.Core.type_obj.html)

static PreComputeRecordConstructor

: recordType:Type \* ?bindingFlags:BindingFlags -> ([obj](Microsoft.FSharp.Core.type_obj.html) [[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html) -> [obj](Microsoft.FSharp.Core.type_obj.html))

static PreComputeRecordReader

: recordType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) \* ?bindingFlags:BindingFlags -> ([obj](Microsoft.FSharp.Core.type_obj.html) -> [obj](Microsoft.FSharp.Core.type_obj.html) [[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html))

static PreComputeTupleConstructor

: tupleType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) -> ([obj](Microsoft.FSharp.Core.type_obj.html) [[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html) -> [obj](Microsoft.FSharp.Core.type_obj.html))

static PreComputeTupleReader

: tupleType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) -> ([obj](Microsoft.FSharp.Core.type_obj.html) -> [obj](Microsoft.FSharp.Core.type_obj.html)[[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html))

static PreComputeTuplePropertyInfo

: tupleType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) \* index:int -> PropertyInfo \* (Type \* int) option

static PreComputeTupleConstructorInfo

: tupleType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) -> ConstructorInfo \* Type option

static PreComputeRecordFieldReader

: info:PropertyInfo \* ?bindingFlags:BindingFlags -> ([obj](Microsoft.FSharp.Core.type_obj.html) -> [obj](Microsoft.FSharp.Core.type_obj.html))

static PreComputeUnionReader

: unionType:UnionCaseInfo \* ?bindingFlags:BindingFlags -> ([obj](Microsoft.FSharp.Core.type_obj.html) -> [obj](Microsoft.FSharp.Core.type_obj.html) [[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html))

static PreComputeUnionTagReader

: unionType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) \* ?bindingFlags:BindingFlags -> ([obj](Microsoft.FSharp.Core.type_obj.html) -> [int](Microsoft.FSharp.Core.type_int.html))

static PreComputeUnionTagMemberInfo

: unionType:[Type](http://msdn2.microsoft.com/en-us/library/System.Type.aspx) \* ?bindingFlags:BindingFlags -> MemberInfo

static PreComputeUnionConstructor

: unionCase:UnionCaseInfo -> ([obj](Microsoft.FSharp.Core.type_obj.html) [[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html) -> [obj](Microsoft.FSharp.Core.type_obj.html))

static PreComputeUnionConstructorInfo

: unionCase:UnionCaseInfo -> ConstructorInfo

## FSharp.Reflection.FSharpType and UnionCase (Static Class)

This static class enables reflection over F# union types, F# record types, F# function types and F# tuple types

Design Criteria

* Create, analyze and manipulate union, record, function and tuple types
* Accesses union cases
* Allow analysis of private record and union types according to a passed-in bindingFlags parameter. No granularity greater than Public/NonPublic is required.

Usage Model

TBD

Performance Criteria

TBD

Signature

[<AbstractClass; Sealed>]

type FSharpType =

static GetRecordFields: recordType:Type \* ?bindingFlags:BindingFlags -> PropertyInfo[]

static GetUnionCases: unionType:Type \* ?bindingFlags:BindingFlags -> UnionCaseInfo[]

static MakeFunctionType: domain:Type \* range:Type -> Type

static MakeTupleType: types:Type[] -> Type

static IsTuple : typ:Type -> bool

static IsFunction : typ:Type -> bool

static IsModule: typ:Type -> bool

static IsRecord: typ:Type \* ?bindingFlags:BindingFlags -> bool

static IsUnion: typ:Type \* ?bindingFlags:BindingFlags -> bool

static GetTupleElements : tupleType:Type -> Type[]

static GetFunctionElements : functionType:Type -> Type \* Type

static MakeFunctionType: Type \* Type -> Type

static MakeTupleType: Type \* [obj](Microsoft.FSharp.Core.type_obj.html)[[]](../fslib/Microsoft.FSharp.Core.type__%5b%5d.html) -> [obj](Microsoft.FSharp.Core.type_obj.html)

type UnionCaseInfo with

member Name : string

member DeclaringType : Type

member GetFields: unit -> PropertyInfo[]

member Tag: int

# FSharp.Quotations (Namespace)

Design Criteria

Quoted expressions annotated decorated with Types and operations related to them. These expressions trees are dynamically annotated with type information which is dynamically checked for consistency.

Alignment Considerations

The first versions of F# quotations were "unaligned". They did not make use of System.Reflection idioms such as System.Type.

The final API is highly aligned with System.Reflection and FSharp.Reflection idioms.

Strong consideration has been given as to whether F# should directly generate LINQ expression trees and/or LINQ statement trees instead of having quotations generate a bespoke expression tree type.

The F# design team has investigated this issue and decided against generating LINQ expression trees on the following grounds

* They do not cover the statement fragment of C#. From the F# perspective this makes them very unsatisfactory and incomplete, e.g. calling a method that returns "void" is not possible
* Several LINQ expression tree nodes are very related to C# and VB language constructs and are a poor match for F#
* LINQ expression tree nodes are extensible (through subclassing) in ways that are highly undesirable in F# practice
* In practice, converting a subset of F# quotations to LINQ expression trees is not that difficult (see the convertor in FSharp.PowerPack)
* The F# design team belive that the longevity of the language depends partly on its hygiene, and the hygiene of its library. Committing to producing LINQ expression trees ties F# very closely to one particular meta-programming technology that is not a good fit for the language.

## FSharp.Quotations.Var, Expr (Type)

Information at the binding site of a variable

[<Sealed>]

type Var =

member Type : Type

member Name : string

new : name:string \* typ:Type \* ?isMutable : bool -> Var

static Global : name:string \* typ:Type -> Var

interface System.IComparable

[<Class>]

type Expr =

member Substitute : substitution:(Var -> Expr option) -> Expr

member GetFreeVars : unit -> seq<Var>

member Type : Type

member CustomAttributes : Expr list

static AddressOf : target:Expr -> Expr

static AddressSet : target:Expr \* value:Expr -> Expr

static Application: functionExpr:Expr \* argument:Expr -> Expr

static Applications: functionExpr:Expr \* arguments:list<list<Expr>> -> Expr

static Call : methodInfo:MethodInfo \* arguments:list<Expr> -> Expr

static Call : obj:Expr \* methodInfo:MethodInfo \* arguments:list<Expr> -> Expr

static Coerce : source:Expr \* target:Type -> Expr

static IfThenElse : guard:Expr \* thenExpr:Expr \* elseExpr:Expr -> Expr

static ForIntegerRangeLoop: loopVariable:Var \* start:Expr \* endExpr:Expr \* body:Expr -> Expr

static FieldGet: fieldInfo:FieldInfo -> Expr

static FieldGet: obj:Expr \* fieldInfo:FieldInfo -> Expr

static FieldSet: fieldInfo:FieldInfo \* value:Expr -> Expr

static FieldSet: obj:Expr \* fieldInfo:FieldInfo \* value:Expr -> Expr

static Lambda : parameter:Var \* body:Expr -> Expr

static Let : letVariable:Var \* letExpr:Expr \* body:Expr -> Expr

static LetRec : bindings:(Var \* Expr) list \* body:Expr -> Expr

static NewObject: constructorInfo:ConstructorInfo \* arguments:Expr list -> Expr

static DefaultValue: expressionType:Type -> Expr

static NewTuple: elements:Expr list -> Expr

static NewRecord: recordType:Type \* elements:Expr list -> Expr

static NewArray: elementType:Type \* elements:Expr list -> Expr

static NewDelegate: delegateType:Type \* parameters:Var list \* body:Expr -> Expr

static NewUnionCase: unionCase:UnionCaseInfo \* arguments:Expr list -> Expr

static PropGet: obj:Expr \* property:PropertyInfo \* ?indexerArgs: Expr list -> Expr

static PropGet: property:PropertyInfo \* ?indexerArgs: Expr list -> Expr

static PropSet: obj:Expr \* property:PropertyInfo \* value:Expr \* ?indexerArgs: Expr list -> Expr

static PropSet: property:PropertyInfo \* value:Expr \* ?indexerArgs: Expr list -> Expr

static Quote: inner:Expr -> Expr

static Sequential: first:Expr \* second:Expr -> Expr

static TryWith: body:Expr \* filterVar:Var \* filterBody:Expr \* catchVar:Var \* catchBody:Expr -> Expr

static TryFinally: body:Expr \* compensation:Expr -> Expr

static TupleGet: tuple:Expr \* index:int -> Expr

static TypeTest: source:Expr \* target:Type -> Expr

static UnionCaseTest: source:Expr \* unionCase:UnionCaseInfo -> Expr

static Value : value:obj \* expressionType:Type -> Expr

static Value : value:'a -> Expr

static Var : variable:Var -> Expr

static VarSet : variable:Var \* value:Expr -> Expr

static WhileLoop : guard:Expr \* body:Expr -> Expr

static Cast : source:Expr -> Expr<'a>

static TryGetReflectedDefinition : methodBase:MethodBase -> Expr option

static Deserialize :

qualifyingType:Type \*

spliceTypes:list<Type> \*

spliceExprs:list<Expr> \*

value:byte[] -> Expr

static RegisterReflectedDefinitions:

assembly:Assembly \*

resource:string \*

serializedValue:byte[] -> unit

static GlobalVar<'a> : name:string -> Expr<'a>

type Expr<'a> =

inherit Expr

## FSharp.Quotations.Patterns (Module)

Design Criteria

The full set of primitive active patterns associated with quoted expression trees.

Usage Model

TBD

Signature

module Patterns =

val (|AddressOf|\_|) : input:Expr -> Expr option

val (|AddressSet|\_|) : input:Expr -> (Expr \* Expr) option

val (|Application|\_|) : input:Expr -> (Expr \* Expr) option

val (|Call|\_|) : input:Expr -> (Expr option \* MethodInfo \* Expr list) option

val (|Coerce|\_|) : input:Expr -> (Expr \* Type) option

val (|FieldGet|\_|) : input:Expr -> (Expr option \* FieldInfo) option

val (|FieldSet|\_|) : input:Expr -> (Expr option \* FieldInfo \* Expr) option

val (|ForIntegerRangeLoop|\_|) : input:Expr -> (Var \* Expr \* Expr \* Expr) option

val (|WhileLoop|\_|) : input:Expr -> (Expr \* Expr) option

val (|IfThenElse|\_|) : input:Expr -> (Expr \* Expr \* Expr) option

val (|Lambda|\_|) : input:Expr -> (Var \* Expr) option

val (|Let|\_|) : input:Expr -> (Var \* Expr \* Expr) option

val (|LetRec|\_|) : input:Expr -> ((Var \* Expr) list \* Expr) option

val (|NewArray|\_|) : input:Expr -> (Type \* Expr list) option

val (|DefaultValue|\_|) : input:Expr -> Type option

val (|NewDelegate|\_|) : input:Expr -> (Type \* Var list \* Expr) option

val (|NewObject|\_|) : input:Expr -> (ConstructorInfo \* Expr list) option

val (|NewRecord|\_|) : input:Expr -> (Type \* Expr list) option

val (|NewUnionCase|\_|) : input:Expr -> (UnionCaseInfo \* Expr list) option

val (|NewTuple|\_|) : input:Expr -> (Expr list) option

val (|PropGet|\_|) : input:Expr -> (Expr option \* PropertyInfo \* Expr list) option

val (|PropSet|\_|) : input:Expr -> (Expr option \* PropertyInfo \* Expr list \* Expr) option

val (|Quote|\_|) : input:Expr -> Expr option

val (|Sequential|\_|) : input:Expr -> (Expr \* Expr) option

val (|TryWith|\_|) : input:Expr -> (Expr \* Var \* Expr \* Var \* Expr) option

val (|TryFinally|\_|) : input:Expr -> (Expr \* Expr) option

val (|TupleGet|\_|) : input:Expr -> (Expr \* int) option

val (|TypeTest|\_|) : input:Expr -> (Expr \* Type) option

val (|UnionCaseTest|\_|) : input:Expr -> (Expr \* UnionCaseInfo) option

val (|Value|\_|) : input:Expr -> (obj \* Type) option

val (|Var|\_|) : input:Expr -> Var option

## FSharp.Quotations.DerivedPatterns (Module)

Design Criteria

Provide useful derived patterns for working with F# quotation trees, based on those common patterns needed to realistically write an interpreter or Quotation-to-LINQ transformer.

Usage Model

TBD

Signature

module DerivedPatterns =

val (|Lambdas|\_|) : input:Expr -> (Var list list \* Expr) option

val (|Applications|\_|) : input:Expr -> (Expr \* Expr list list) option

val (|AndAlso|\_|) : input:Expr -> (Expr \* Expr) option

val (|OrElse|\_|) : input:Expr -> (Expr \* Expr) option

val (|Unit|\_|) : input:Expr -> unit option

val (|Bool|\_|) : input:Expr -> bool option

val (|String|\_|) : input:Expr -> string option

val (|Single|\_|) : input:Expr -> float32 option

val (|Double|\_|) : input:Expr -> float option

val (|Char|\_|) : input:Expr -> char option

val (|SByte|\_|) : input:Expr -> sbyte option

val (|Byte|\_|) : input:Expr -> byte option

val (|Int16|\_|) : input:Expr -> int16 option

val (|UInt16|\_|) : input:Expr -> uint16 option

val (|Int32|\_|) : input:Expr -> int32 option

val (|UInt32|\_|) : input:Expr -> uint32 option

val (|Int64|\_|) : input:Expr -> int64 option

val (|UInt64|\_|) : input:Expr -> uint64 option

val (|SpecificCall|\_|) : templateParameter:Expr -> (Expr -> (list<Type> \* list<Expr>) option)

val (|MethodWithReflectedDefinition|\_|) : methodBase:MethodBase -> Expr option

val (|PropertyGetterWithReflectedDefinition|\_|) : propertyInfo:PropertyInfo -> Expr option

val (|PropertySetterWithReflectedDefinition|\_|) : propertyInfo:PropertyInfo -> Expr option

## FSharp.Quotations.ExprShape (Module)

Design Criteria

Allow generic traversal of F# quotation trees with respect to the basic shape of the tree including its "binding structure".

Usage Model

TBD

Performance Criteria

TBD

Example

The following example shows a traversal that raises an exception if the tree contains a use of a variable called "foo"

open ExprShape

let rec rw t =

match t with

| ShapeVar(v) -> if v.Name = "foo" then failwith "foo" else t

| ShapeCombination(obj,args) -> RebuildShapeCombination(obj,List.map rw args)

| ShapeLambda(v,arg) -> Expr.Lambda(v,rw arg)

Signature

module ExprShape =

val (|ShapeVar|ShapeLambda|ShapeCombination|) :

input:Expr -> Choice<Var, (Var \* Expr), (obj \* list<Expr>)>

val RebuildShapeCombination : shape:obj \* arguments:list<Expr> -> Expr

# FSharp.Text (Namespace)

## FSharp.Text.Format (Type)

Typesafe printf format strings, with phantom type annotations added by the F# compiler

Design Criteria

TBD

Usage Model

TBD

Signature

TBD

// This carries extra information inserted by the compiler

type Format<'T,'U,'V,'d> =

new : string -> Format<'T,'U,'V,'d>

member Value : string

// This carries extra information.

type Format<'T,'U,'V,'d,'e> =

new : string -> Format<'T,'U,'V,'d,'e>

inherit Format<'T,'U,'V,'d>

## FSharp.Text.Printf (Module)

Design Criteria

Extensible typesafe printf-style formatting for numbers and other datatypes.

Naming conention

Suffix n = newline

Prefix b = output intermediate results via StringBuilder

Prefix e = output to stderr

Prefix f = output to TextWriter

Prefix tw = output to #TextWriter

Prefix s = generate a string

Prefix k = pass the generated value to a function, called immediately at the end of printing

Usage Model

TBD

Signature

type BuilderFormat = Format<'T,StringBuilder,unit,'d>

type BuilderFormat = BuilderFormat<'T,unit>

type BuildStringFormat = BuilderFormat<'T,string>

type StringFormat = Format<'T,unit,string,'d>

type StringFormat = StringFormat<'T,string>

type TextWriterFormat = Format<'T,TextWriter,unit,'d>

type TextWriterFormat = TextWriterFormat<'T,unit>

val bfprintf: TextWriter -> BuilderFormat<'T> -> 'T

val bprintf: StringBuilder -> BuilderFormat<'T> -> 'T

val bsprintf: BuildStringFormat<'T> -> 'T

val eprintfn: TextWriterFormat<'T> -> 'T

val failwithf: StringFormat<'T,'d> -> 'T

val fprintf: TextWriter -> TextWriterFormat<'T> -> 'T

val fprintfn: TextWriter -> TextWriterFormat<'T> -> 'T

val kbprintf: (unit -> 'd) -> StringBuilder -> BuilderFormat<'T,'d> -> 'T

val kbsprintf: (string -> 'd) -> BuilderFormat<'T,'d> -> 'T

val kfprintf: (unit -> 'd) -> TextWriter -> TextWriterFormat<'T,'d> -> 'T

val kprintf: (string -> 'd) -> StringFormat<'T,'d> -> 'T

val ksprintf: (string -> 'd) -> StringFormat<'T,'d> -> 'T

val ktwprintf: (unit -> 'd) -> TextWriter -> TextWriterFormat<'T,'d> -> 'T

val printf: TextWriterFormat<'T> -> 'T

val printfn: TextWriterFormat<'T> -> 'T

val sprintf: StringFormat<'T> -> 'T

# FSharp.NativeInterop (Namespace)

## FSharp.NativeInterop.NativePtr (Module)

Inlined unsafe primitives to enable direct pointer manipulations from F# programs.

Design Criteria

* Enable fidelity with C# unsafe programming

Performance Criteria

TBD

Usage Model

TBD

Signature

module NativePtr =

val add: nativeptr<'T> -> [int](Microsoft.FSharp.Core.type_int.html) -> nativeptr<'T>

val ofNativeInt: [nativeint](Microsoft.FSharp.Core.type_nativeint.html) -> nativeptr<'T>

val set: nativeptr<'T> -> [int](Microsoft.FSharp.Core.type_int.html) -> 'T -> [unit](Microsoft.FSharp.Core.type_unit.html)

val stackalloc<'T>: unit -> nativeptr<'T>

val toNativeInt: nativeptr<'T> -> nativeint

val write: nativeptr<'T> -> 'T -> unit

# Language Primitives

## FSharp.Core.FastFunc (Type)

A class-based function type with efficient implementation of invocation of curried functions (int -> int -> int).

Design Criteria

* opaque function type with a single Invoke method
* efficient curried invocation for up to 4 arguments simultaneously
* possible to create values from C#

Alignment

The type has similarity to System.Func<T,U>. However that type has intrinsically very poor performance for curried invocations.

Future-proofing

F# made an (unactioned) CLR ask in Dev10 to have a design change to delegates that would allow for efficient curried invocation. If this design change were ever actioned then F# would be willing to take a breaking binary change (and breaking change to how F# function values are accessed from C#) in order to align FastFunc and Func.

In the meantime no specific action has been taken for alignment here.

The .NET type used to represent F# function values. This type is not typically used directly from F# code, though may be used from other .NET languages.

type FastFunc<'T,'U> =

abstract member Invoke: 'T -> 'U

static op\_Implicit: Converter<'T,'U> -> ('T -> 'U)

static op\_Implicit: ('T -> 'U) -> Converter<'T,'U>

WW

WXWX

static InvokeFast5: [FastFunc](Microsoft.FSharp.Core.type_FastFunc.html)<'T,('U -> 'V -> 'W -> 'X -> 'Y)> \* 'T \* 'U \* 'V \* 'W \* 'X -> 'Y

new: [unit](Microsoft.FSharp.Core.type_unit.html) -> [FastFunc](Microsoft.FSharp.Core.type_FastFunc.html)<'T,'U>

## Tuple (functional tuples)

Compiled versions of F# tuple types. These are a set ot types implementing immutable, strongly-typed tuples with structural comparison and hash semantics. These are not used directly from F# code, though these compiled forms are seen by other .NET languages.

Alignment Considerations

A shared tuple type is available in .NET 4.0 and is used by F#.

F# is very performance-sensitive to its implementation of tuples. The following were essential F# criteria here before it was agreed to use a shared tuple type:

* Immutability (to allow compiler optimizations)
* Sealed (to allow compiler optimizations)
* Minimalist API (to reduce per-instantiation costs)
* Structural comparison/hashing semantics

Sample Signature

## FSharp.Core.TypeFunc (Type)

The .NET type used to represent F# first-class type function values. This type is not typically used directly from F# code.

[<AbstractClass>]

type FSharpTypeFunc =

abstract Specialize<'T> : unit -> obj

## FSharp.Core.Choice (Type)

Design Criteria

TBD

Performance Criteria

TBD

Usage Model

N/A

Signature

type Choice<'T,'U> =

| Choice1Of2 of 'T

| Choice2Of2 of 'U

etc.

## FSharp.Core.LanguagePrimitives (Module)

Design Criteria

Calls to these primitives are emitted by the F# compiler

Performance Criteria

TBD

Usage Model

N/A

Signature

module LanguagePrimitives =

val GenericEquality : e1:'a -> e2:'a -> bool

val GenericComparison : e1:'a -> e2:'a -> int

val GenericComparisonBimodal : e1:'a -> e2:'a -> int

val GenericLessThan : e1:'a -> e2:'a -> bool

val GenericGreaterThan : e1:'a -> e2:'a -> bool

val GenericLessOrEqual : e1:'a -> e2:'a -> bool

val GenericGreaterOrEqual : e1:'a -> e2:'a -> bool

val GenericMinimum : e1:'a -> e2:'a -> 'a

val GenericMaximum : e1:'a -> e2:'a -> 'a

val PhysicalEquality : e1:'a -> e2:'a -> bool

val PhysicalHash : obj:'a -> int

val GenericHash : obj:'a -> int

val DefaultValue<'a when 'a : null> : 'a

val EnumOfValue : value:'u -> 'e when 'e : enum<'u>

val EnumToValue : enum:'e -> 'u when 'e : enum<'u>

val ParseInt32 : s:string -> int32

val ParseUInt32 : s:string -> uint32

val ParseInt64 : s:string -> int64

val ParseUInt64 : s:string -> uint64

val GenericZeroDynamic : unit -> 'a

val GenericOneDynamic : unit -> 'a

val AdditionDynamic : x:'a -> y:'b -> 'c

val CheckedAdditionDynamic : x:'a -> y:'b -> 'c

val MultiplyDynamic : x:'a -> y:'b -> 'c

val CheckedMultiplyDynamic : x:'a -> y:'b -> 'c

val DivideByIntDynamic : x:'a -> y:int -> 'a

val GenericZero< ^a > : ^a when ^a : (static member Zero : ^a)

val GenericOne< ^a > : ^a when ^a : (static member One : ^a)

val DivideByInt< ^a > : x:^a -> y:int -> ^a when ^a : (static member DivideByInt : ^a \* int -> ^a)

/// Compiler-recognized operators

module IntrinsicOperators =

val ( & ) : e1:bool -> e2:bool -> bool

val ( && ) : e1:bool -> e2:bool -> bool

val ( or ) : e1:bool -> e2:bool -> bool

val ( || ) : e1:bool -> e2:bool -> bool

val ( ~& ) : obj:'a -> 'a byref

val ( ~&& ) : obj:'a -> nativeptr<'a>

/// Compiler-generated primitives

module IntrinsicFunctions =

val UnboxGeneric<'a> : obj -> 'a

val UnboxFast<'a> : obj -> 'a

val TypeTestGeneric<'a> : obj -> bool

val TypeTestFast<'a> : obj -> bool

val ArrayGet : 'a array -> int -> 'a

val CreateInstance : unit -> 'a when 'a : (new : unit -> 'a)

val MakeDecimal : lo:int \* med:int \* hi:int \* isNegative:bool \* scale:byte -> decimal

module HashCompare =

val PhysicalHashIntrinsic : 'a -> int

val PhysicalEqualityIntrinsic : x:'a -> y:'a -> bool

val GenericHashParamIntrinsic : 'a -> int byref -> int

val GenericComparisonIntrinsicBimodal : x:'a -> y:'a -> int

val GenericComparisonIntrinsic : x:'a -> y:'a -> int

val GenericEqualityIntrinsic : x:'a -> y:'a -> bool

val GenericLessThanIntrinsic : x:'a -> y:'a -> bool

val GenericGreaterThanIntrinsic : x:'a -> y:'a -> bool

val GenericGreaterOrEqualIntrinsic : x:'a -> y:'a -> bool

val GenericLessOrEqualIntrinsic : x:'a -> y:'a -> bool

## FSharp.Core.OptimizedRanges (Module)

Design Criteria

Support the efficient generation of IEnumerables for F# range expressions

Performance Criteria

TBD

Usage Model

N/A

Signature

module OptimizedRanges =

val RangeByte: [byte](Microsoft.FSharp.Core.type_byte.html) -> [byte](Microsoft.FSharp.Core.type_byte.html) -> [byte](Microsoft.FSharp.Core.type_byte.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[byte](Microsoft.FSharp.Core.type_byte.html)>

val RangeChar: [char](Microsoft.FSharp.Core.type_char.html) -> [char](Microsoft.FSharp.Core.type_char.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[char](Microsoft.FSharp.Core.type_char.html)>

val RangeDouble: [float](Microsoft.FSharp.Core.type_float.html) -> [float](Microsoft.FSharp.Core.type_float.html) -> [float](Microsoft.FSharp.Core.type_float.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[float](Microsoft.FSharp.Core.type_float.html)>

val RangeSingle: [float32](Microsoft.FSharp.Core.type_float32.html) -> [float32](Microsoft.FSharp.Core.type_float32.html) -> [float32](Microsoft.FSharp.Core.type_float32.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[float32](Microsoft.FSharp.Core.type_float32.html)>

val RangeGeneric: 'T -> ('T -> 'T -> 'T) -> 'T -> 'T -> 'T -> [seq](Microsoft.FSharp.Core.type_seq.html)<'T>

val RangeInt32: [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> [int](Microsoft.FSharp.Core.type_int.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[int](Microsoft.FSharp.Core.type_int.html)>

val RangeInt16: [int16](Microsoft.FSharp.Core.type_int16.html) -> [int16](Microsoft.FSharp.Core.type_int16.html) -> [int16](Microsoft.FSharp.Core.type_int16.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[int16](Microsoft.FSharp.Core.type_int16.html)>

val RangeInt64: [int64](Microsoft.FSharp.Core.type_int64.html) -> [int64](Microsoft.FSharp.Core.type_int64.html) -> [int64](Microsoft.FSharp.Core.type_int64.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[int64](Microsoft.FSharp.Core.type_int64.html)>

val RangeIntPtr: [nativeint](Microsoft.FSharp.Core.type_nativeint.html) -> [nativeint](Microsoft.FSharp.Core.type_nativeint.html) -> [nativeint](Microsoft.FSharp.Core.type_nativeint.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[nativeint](Microsoft.FSharp.Core.type_nativeint.html)>

val RangeSByte: [sbyte](Microsoft.FSharp.Core.type_sbyte.html) -> [sbyte](Microsoft.FSharp.Core.type_sbyte.html) -> [sbyte](Microsoft.FSharp.Core.type_sbyte.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[sbyte](Microsoft.FSharp.Core.type_sbyte.html)>

val RangeUInt16: [uint16](Microsoft.FSharp.Core.type_uint16.html) -> [uint16](Microsoft.FSharp.Core.type_uint16.html) -> [uint16](Microsoft.FSharp.Core.type_uint16.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[uint16](Microsoft.FSharp.Core.type_uint16.html)>

val RangeUInt32: [uint32](Microsoft.FSharp.Core.type_uint32.html) -> [uint32](Microsoft.FSharp.Core.type_uint32.html) -> [uint32](Microsoft.FSharp.Core.type_uint32.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[uint32](Microsoft.FSharp.Core.type_uint32.html)>

val RangeUInt64: [uint64](Microsoft.FSharp.Core.type_uint64.html) -> [uint64](Microsoft.FSharp.Core.type_uint64.html) -> [uint64](Microsoft.FSharp.Core.type_uint64.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[uint64](Microsoft.FSharp.Core.type_uint64.html)>

val RangeUIntPtr: [unativeint](Microsoft.FSharp.Core.type_unativeint.html) -> [unativeint](Microsoft.FSharp.Core.type_unativeint.html) -> [unativeint](Microsoft.FSharp.Core.type_unativeint.html) -> [seq](Microsoft.FSharp.Core.type_seq.html)<[unativeint](Microsoft.FSharp.Core.type_unativeint.html)>

also AbsDynamic and other dynamic invocation primitives

## FSharp.Core. OptimizedClosures (Module)

Design Criteria

The .NET type used to represent F# function values that accept multiple curried arguments without intervening execution. These types should not typically used directly from either F# code or from other .NET languages except when hand-optimizing inner loops.

Performance Criteria

TBD

Usage Model

TBD

Signature

[<AbstractClass>]

type FastFunc2<'T,'U,'V> =

inherit FastFunc <'T,('U -> 'V)>

abstract member Invoke : arg1:'T \* arg2:'U -> 'V

static member Adapt : ('T -> 'U -> 'V) -> FastFunc2<'T,'U,'V>

new : unit -> FastFunc2 <'T,'U,'V>

[<AbstractClass>]

type FastFunc3<'T,'U,'V,'W> =

inherit FastFunc <'T,('U -> 'V -> 'W)>

abstract member Invoke : arg1:'T \* arg2:'U \* arg3:'V -> 'W

static member Adapt : ('T -> 'U -> 'V -> 'W) -> FastFunc3<'T,'U,'V,'W>

new : unit -> FastFunc3 <'T,'U,'V,'W>

[<AbstractClass>]

type FastFunc4<'T,'U,'V,'W,'X> =

inherit FastFunc <'T,('U -> 'V -> 'W -> 'X)>

abstract member Invoke : arg1:'T \* arg2:'U \* arg3:'V \* arg4:'W -> 'X

static member Adapt : ('T -> 'U -> 'V -> 'W -> 'X) -> FastFunc4<'T,'U,'V,'W,'X>

new : unit -> FastFunc4 <'T,'U,'V,'W,'X>

[<AbstractClass>]

type FastFunc5<'T,'U,'V,'W,'X,'Y> =

inherit FastFunc <'T,('U -> 'V -> 'W -> 'X -> 'Y)>

abstract member Invoke : arg1:'T \* arg2:'U \* arg3:'V \* arg4:'W \* arg5:'X -> 'Y

static member Adapt : ('T -> 'U -> 'V -> 'W -> 'X -> 'Y) -> FastFunc5<'T,'U,'V,'W,'X,'Y>

new : unit -> FastFunc5 <'T,'U,'V,'W,'X,'Y>

## FSharp.Core.Collecitons.SequenceExpressionHelpers (Module)

Design Criteria

Support the compilation of F# sequence expressions

Performance Criteria

TBD

Usage Model

N/A

Signature

module SequenceExpressionHelpers =

val EnumerateWhile : guard:(unit -> bool) -> sequence:seq<'a> -> seq<'a>

val EnumerateThenFinally : sequence:seq<'a> -> compensation:(unit -> unit) -> seq<'a>

val EnumerateFromFunctions: create:(unit -> 'a) -> moveNext:('a -> bool) -> current:('a -> 'b) -> seq<'b>

val EnumerateUsing : resource:'a -> sequence:('a -> #seq<'b>) -> seq<'b> when 'a :> IDisposable

val Generate : opener:(unit -> 'b) -> generator:('b -> 'a option) -> closer:('b -> unit) -> seq<'a>

val GenerateUsing : opener:(unit -> ('a :> IDisposable)) -> generator:('a -> 'b option) -> seq<'b>

## FSharp.Collections.ComparisonIdentity (Module)

Design Criteria

Permit the efficient conversion between IComparer and F# curried comparison functions ('T -> 'T -> int)

Usage Model

N/A

module ComparisonIdentity =

val Structural<'Key> : IComparer<'Key>

val GetFastComparisonFunction : comparer:IComparer<'a> -> FastFunc2<'a,'a,int>

val GetFastStructuralComparisonFunction : unit -> FastFunc2<'a,'a,int>

val FromFunction : comparer:('Key -> 'Key -> int) -> IComparer<'Key>

## FSharp.Collections.HashIdentity (Module)

Design Criteria

Permit the efficient conversion between IEqualityComparer and F# pairs of hashing and curried equality functions ('T -> 'T -> bool)

Usage Model

N/A

module HashIdentity =

val Structural<'Key> : IEqualityComparer<'Key>

val Reference<'Key> : IEqualityComparer<'Key>

val FromFunctions<'Key> : ('Key->int) -> ('Key->'Key->bool) -> IEqualityComparer<'Key>

## FSharp.Core.FuncConvert (Type)

Design Criteria

Support the conversion of CodeDom delegate creations for F# code

Usage Model

N/A

type FuncConvert =

static ToFastFunc: Converter<'T,'U> -> ('T -> 'U)

static ToFastFunc: Action<'T> -> ('T -> [unit](Microsoft.FSharp.Core.type_unit.html))

static FuncFromTupled:

('T1 \* 'T2 \* 'T3 \* 'T4 \* 'T5 -> 'U) -> ('T1 -> 'T2 -> 'T3 -> 'T4 -> 'T5 -> 'U)

static FuncFromTupled:

('T1 \* 'T2 \* 'T3 \* 'T4 -> 'U) -> ('T1 -> 'T2 -> 'T3 -> 'T4 -> 'U)

static FuncFromTupled:

('T1 \* 'T2 \* 'T3 -> 'U) -> ('T1 -> 'T2 -> 'T3 -> 'U)

static FuncFromTupled:

('T1 \* 'T2 -> 'U) -> ('T1 -> 'T2 -> 'U)

static FuncFromTupled:

('T -> 'U) -> ('T -> 'U)

## FSharp.Core.Unit (Type)

The type 'unit', which has only one literal value "()", always represented as "null"

Design Criteria

Support the "unit" type and literal in F# programming

Usage Model

N/A

type Unit

type unit = Unit