F# Technical Notes

[Introduction 4](#_Toc388370376)

[1 Language Design Notes & FAQs 5](#_Toc388370377)

[1.1 Grammar 5](#_Toc388370378)

[1.2 Type Inference 5](#_Toc388370379)

[1.2.1 Erasure of “’a :> obj” constraints 5](#_Toc388370380)

[1.2.2 Detailed Notes for Subsumption, Condensation and Decondensation 5](#_Toc388370381)

[1.2.3 Detailed Notes for Inference for Mutually Recursive Classes 7](#_Toc388370382)

[1.2.4 Value Restriction Issues 10](#_Toc388370383)

[1.2.5 Using Type Names as Constructor Functions 10](#_Toc388370384)

[1.2.6 Auto-Open of Own Namespace 11](#_Toc388370385)

[1.2.7 Implicit Resolution of Operators to Static Member Constraints 11](#_Toc388370386)

[1.2.8 Warning 1182 (unused variables) 11](#_Toc388370387)

[1.3 Expressions 12](#_Toc388370388)

[1.3.1 Design Notes for Ranges over Enums 12](#_Toc388370389)

[1.4 Object Model 12](#_Toc388370390)

[1.4.1 The Explicit Class Syntax 12](#_Toc388370391)

[1.4.2 Required “this” and “static” in members 12](#_Toc388370392)

[1.4.3 Design Notes for Interface Implementation Expressions 13](#_Toc388370393)

[1.4.4 Structural Types 14](#_Toc388370394)

[1.4.5 Immediately Recursive Types 15](#_Toc388370395)

[1.5 Methods and Functions 16](#_Toc388370396)

[1.5.1 “mutable” in argument position 16](#_Toc388370397)

[1.6 Structs 16](#_Toc388370398)

[1.6.1 “let” in structs 16](#_Toc388370399)

[1.7 Sequence and Computation Expressions 17](#_Toc388370400)

[1.7.1 History of Incomplete matches in “for” loop patterns 17](#_Toc388370401)

[1.8 Units of Measure 17](#_Toc388370402)

[1.8.1 Systematic Design Patterns and Overloading for Unit Conversion 17](#_Toc388370403)

[1.9 Scripting 18](#_Toc388370404)

[1.9.1 Should Scripts be “Source Code Assemblies”? 18](#_Toc388370405)

[1.10 F# Interactive 19](#_Toc388370406)

[1.10.1 F# Interactive and AssemblyResolve Event 19](#_Toc388370407)

[1.10.2 Known Limitation: F# Interactive & IIS/WCF 19](#_Toc388370408)

[1.10.3 Known Limitation: F# Interactive & Transportable Code 19](#_Toc388370409)

[2 FSharp.Core Design Notes & FAQs 21](#_Toc388370410)

[2.1 Basic Operators 21](#_Toc388370411)

[2.1.1 “enum” only works on 32-bit 21](#_Toc388370412)

[2.1.2 Why both “\*\*” and “pown” 21](#_Toc388370413)

[2.1.3 Dynamic Evaluation of Basic Operators 22](#_Toc388370414)

[2.2 List, Array, Option, Seq 23](#_Toc388370415)

[2.2.1 Should List.sum etc. be checked or unchecked? 23](#_Toc388370416)

[2.3 Microsoft.FSharp.Reflection 23](#_Toc388370417)

[2.3.1 Why do we need “BindingFlags”? 23](#_Toc388370418)

[2.4 Microsoft.FSharp.Control 23](#_Toc388370419)

[2.4.1 F# Async Design Overview and Principles 23](#_Toc388370420)

[2.4.2 Async “Stay In Context” 25](#_Toc388370421)

[2.4.3 Trampolines 26](#_Toc388370422)

[2.4.4 Bind hijacking check 27](#_Toc388370423)

[2.4.5 Splitting up of AsyncParams 27](#_Toc388370424)

[2.5 Microsoft.FSharp.Linq 29](#_Toc388370425)

[2.5.1 Query design 29](#_Toc388370426)

[2.5.2 Equality and comparison in queries 29](#_Toc388370427)

[3 Library-only Language Features 30](#_Toc388370428)

[3.1 Static Optimizations 30](#_Toc388370429)

[3.2 Mutating tail-cons of lists 30](#_Toc388370430)

[3.3 Embedded IL 31](#_Toc388370431)

[4 Binary Formats (Notes) 33](#_Toc388370432)

[5 F# and Serialization 34](#_Toc388370433)

[5.1.1 Requirement: Relevant Serializers 34](#_Toc388370434)

[5.1.2 Requirement: Stability of serialization formats 34](#_Toc388370435)

[5.1.3 Requirement: Interaction with Partial Trust and Security Transparency 34](#_Toc388370436)

[5.1.4 Requirement: Stability across platforms 35](#_Toc388370437)

[5.2 Overview of Serializers 35](#_Toc388370438)

[5.2.1 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 35](#_Toc388370439)

[5.2.2 System. Runtime.Serialization.DataContractSerialization and System. Runtime.Serialization.Json.DataContractJsonSerializer 35](#_Toc388370440)

[5.2.3 System.Runtime.Serialization.Formatters.Soap.SoapFormatter 36](#_Toc388370441)

[5.2.4 System.Xml.Serialization.XmlSerializer 36](#_Toc388370442)

[5.3 Overview of Serialization Programming Models 37](#_Toc388370443)

[5.4 F#-Related Values to Consider w.r.t. Serialization 38](#_Toc388370444)

[5.5 Walkthrough for F# Serialization to JSON 39](#_Toc388370445)

[5.6 Sample JSON Wire Formats 44](#_Toc388370446)

[6 Assembly Reference Resolution 46](#_Toc388370447)

[6.1 Compiler Search Locations 46](#_Toc388370448)

[6.2 Interactive Search Locations 46](#_Toc388370449)

[6.3 F# Attributes 47](#_Toc388370450)

[6.3.1 OCamlCompatibility 47](#_Toc388370451)

[7 Compiler Architectural Breakdown 48](#_Toc388370452)

[7.1 Miscellaneous Architectural Questions 51](#_Toc388370453)

[8 Compiler Performance Analysis 52](#_Toc388370454)

[8.1 Methodology: 52](#_Toc388370455)

[8.2 Analyzing Results 52](#_Toc388370456)

[8.3 Primary Phases 52](#_Toc388370457)

[8.4 Parsing/Lexing phase 53](#_Toc388370458)

[8.5 Typecheck phase 53](#_Toc388370459)

[8.5.1 Typecheck phase - algorithmic aspects 53](#_Toc388370460)

[8.5.2 Typecheck phase - representation aspects 53](#_Toc388370461)

[8.5.3 Typecheck phase – low level and CLR costs 54](#_Toc388370462)

[8.6 TAST 🡪 ILX phase 55](#_Toc388370463)

[8.7 ILX 🡪 IL phase 55](#_Toc388370464)

[8.8 Write Binary phase 55](#_Toc388370465)

[8.8.1 Write PDB 55](#_Toc388370466)

[8.9 Memory Usage of Compiler and Language Service 55](#_Toc388370467)

[8.9.1 Taking a memory dump using windbg and SOS profiling 56](#_Toc388370468)

[8.9.2 Top Entries By Type When Typechecking 57](#_Toc388370469)

[8.9.3 Example dump during ILX Generation 58](#_Toc388370470)

[9 Coding Conventions in the F# Compiler 61](#_Toc388370471)

[9.1 Abbreviations used in the F# Compiler Code 61](#_Toc388370472)

[10 F# Code: Optimized v. Debug 66](#_Toc388370473)

[10.1 Interaction with Debug Tools 66](#_Toc388370474)

[10.1.1 Just My Code, Tailcalls and Stepping (VS2008) 66](#_Toc388370475)

[10.2 Debug v. Optimized Code Generation 66](#_Toc388370476)

[11 FsYacc Notes 68](#_Toc388370477)

[11.1 Getting better error messages out of fsyacc generated parsers 68](#_Toc388370478)

[12 History of the F# Compiler Codebase and Project 69](#_Toc388370479)

[13 F# Async Ideas 72](#_Toc388370480)

[13.1 Async State Machines 72](#_Toc388370481)

[13.2 AsyncResultCell 72](#_Toc388370482)

[13.3 Task Interop 72](#_Toc388370483)

[13.4 IAsyncDisposable 72](#_Toc388370484)

[13.5 ThreadAffinityContext 74](#_Toc388370485)

[13.6 AsyncSeq 75](#_Toc388370486)

[13.7 Making MailboxProcessor MarshalByRef 79](#_Toc388370487)

[13.8 Enabling async hops across domains 79](#_Toc388370488)

# Introduction

This document contains a collection of internal technical information related to the F# project.

Some portions may eventually be published as external specifications, notes or blog entries.

The purpose is to capture the critical parts of email discussions, design notes, FAQs and other material in one place, to act as

* a resource for new team members
* a resource for future design decisions

The document is a bit of a dumping ground and can flux – try to keep it neat & tidy, but if needed delete old/misleading material, remove material where there is a better source, and update material to represent latest information.

# Language Design Notes & FAQs

## Grammar

The note below captures an attempt to produce a “simplified grammar” from the parser implementation.

* *remove all non-#light rules*
* *remove all rules containing "recover" or "error"*
* *remove deprecated and library-only constructs*

## Type Inference

### Erasure of “’a :> obj” constraints

F# type inferences erases these constraints. Here’s a relevant design note:

*Here is one of the ramifications erasing “'a :> obj” constraints.*

type Stack<'a>(n)  =

    let mutable contents = Array.zeroCreate<'a>(n)

    ...

    member buf.PrintStack() =

        for i = 0 to (count - 1) do

            System.Console.Write("{0}",contents.[i])

*In particular, the declared type parameter 'a doesn’t declare a constraint 'a :> obj. This means that passing a value of type 'a to System.Console.WriteLine(string,obj) is not permitted (since the implied subsumption constraint is not solvable). As a result I’d recommend we continue to erase 'a :> obj constraints and live with the slight inconsistency that gives w.r.t. condensation.*

*One way to look at this is that erasing this constraint is what enables us to say “everything in F# is an object” and have that work in practice.*

### Detailed Notes for Subsumption, Condensation and Decondensation

Some notes on the chosen design for subsumption/condensation/decondensation. During generalization, signatures inferred to be of the form

F : … #A … -> R

Are “condensed” to

F : … A … -> R

Where arguments are determined by looking at the de-curried, de-tupled view of the type of F. Condensation doesn’t apply to arguments of unconstrained variable type, e.g.

F: 'a -> unit

Given a reference a function or member F, and the de-curried, de-tupled view of the type of F uninstantiated function/member signature is of the form

F : … A … -> R

Then the reference is “de-condensed”, i.e. interpreted as a function

(fun … arg … -> F … (arg :> A) …)

Notes:

* This applies regardless of whether the signature of F was generated via condensation. For example, it applies to

let f (x:obj) = 1

* De-condensation is determined **after** an explicit instantiation is taken into account but before any arguments are checked, e.g. these **do** typecheck

let M<'b>(c:'b, d:'b) = 1

let obj = new obj()

let str = ""

M<obj>(obj,str)

M<obj>(str,obj)

M<obj>(obj,obj)

M<obj>(str,str)

M(obj,obj)

M(str,str)

But these do NOT (because the target type is variable, not nominal)

M(obj,str)

M(str,obj)

* Some special rules apply when determining the exact elaborated form w.r.t. quotations. (they are the hardest part to implement)
* De-condensation applies in a similar fashion to applications of data contructors (though not tuples, which have variable argument types prior to instantiation)

The philosophy of this design is very much in line with F# in general. Basically, flexibility (aka polymorphism via type parameters) is

* captured at let-bindings and other declarations (through generalization)
* released at uses of the items named in let-bindings (through instantiation and de-condensation)

With the added dimension:

* Flexibility stems from the presence of nominal types in signatures of **named** functions, **named** data constructors and **named** methods, and also from **user**-**specified** type instantiations when applying these.
* It doesn’t come from the application of **arbitrary** function expressions (just as (fun x -> x) is NOT generic in F# - it is only generic if you give it a name). That is, we don’t support subsumption at function application.
* It doesn’t come from **inferred** type instantiations

#### Alternative designs:

Some notes on the rational for not adopting a “fully type-directed” approach to subsumption...

A fully type-directed approach to subsumption violates the principle of “let-substitutability” , e.g. consider the following

type C() =

    member x.P = 1

type D() =

    inherit C()

    member x.P2 = 1

let SomeFunction (c1:C list) = 1

Then under the proposal

SomeFunction [new D(); new D()]

Can’t be converted to

let inps = [new D(); new D()]

SomeFunction inps

This is a routine, normal transformation to make for an F# program. One way to look at this is “could you answer a forum post where someone asked ‘why can’t I replace the first with the second”. Most people say they would have little or no chance of explaining this, except perhaps to a C# 3.0 programmer. But not, say, to a Python programmer. Any explanation which has to effectively say “well, inps gets this type and it can’t be used at this other type unless you expand it as a list literal” just won’t fly to many prospective users.

Substitutability properties of the kind above are absolutely fundamental to F# (more so than C# 3.0), and the syntax of things such as list and tuple literals are really deeply biased towards assuming that these properties hold. There are, of course, a number of small ways in which we lose substitutability for F# programs, however we have so far been careful to limit this.

As a result we ended up adopting a much more mild proposal where we keep the “condensation” rule and reverse it on each use of a function. This means

* subsumption would apply to uses of functions, data cosntructors, record mutators etc. based on the use of nominal types in their uninstantiated declared signatures
* subsumption would not apply at applications of “generic” functions and data constructors (such lists)

This gave the uniformity needed between members and functions.

### Detailed Notes for Inference for Mutually Recursive Classes

This design note is based on the original documentation of the v2.0 design for "better inference for mutually recursrive generic classes".

This problem is about how recursive type inference leads to over-constrained types for functions and member. In particular, prior to F# v2.0, generalization of member bindings is not applied until the end of a recursive group.  This describes a design change which improved this for v2.0. The rule itself is in some ways imperfect, but it copes with many common practical cases of a recurring problem with F# type inference in the context of object-oriented programming.

The problem of over-constrained recursive inference manifests itself in very simple circumstances, e.g.

module Example1 =

    type C() =

        member x.Identity inp = inp

        member x.Q1 = x.Identity 3 // ok, but constrains Identity to work on integers

        member x.Q2 = x.Identity "3" // error – int <> string

module Example2 =

    type C<'T>() =

        member x.P = 1

        member x.Q = (new C<int>()).P  // error – 'T is constrained to be int

Here "P" is being used as both C<'T>.P and C<int>.P within a recursive scope, and an error is given because C.P has not been generalized prior to the point of use.

Now, we already have an early generalization rule for cases where a full type signature is given for a member, e.g. adding "x.P : int" is sufficient to allow the generic use of P. While useful as a mechanism of last resort, this is not an intuitive rule.  In particular, it is common that, after they are checked, members have inferred types that are completely independent of the remaining type inference for remaining members. In these cases members can be generalized as soon as they are checked.

The design change was to add an additional early generalization rule as follows

* After checking each binding (i.e. member) in a recursive group, all un-generalized, checked bindings are analyzed to see which can now be generalized
* A binding can be generalized if its inferred type is closed w.r.t. any inference variables present in the types of members which are not yet checked or which, in turn, can not be generalized. This is an iterative (greatest fixed point) computation where we repeatedly remove bindings from the set of un-generalized, checked bindings until a stable set of generalizable bindings remain.
  + That is, once checked bindings can be generalized early if their types are independent of any inference type variables present in the partially inferred types of unchecked members.
* Bindings which are generalized can be used generically later in the recursive group.

For example, consider the following case:

type Vec<'a>(x:'a,y:'a) =

  member v.x = x

  member v.y = y

  member v.Mul(w:Vec<'b>) : Vec<'a\*'b> = new Vec<'a\*'b>((v.x,w.x),(v.y,w.y))

After we check the bindings  for v.X and v.Y we know the full type signatures for these bindings and we can easily see that these types are completely independent of the type of v.Mul. Hence the bindings can each be generalized immediately after they are checked.

Likewise, the rule also applies to regulare "let rec" bindings

    let rec f x = x

    and g x = f 1

Here "f" can be generalized early. This rule has several nice properties

* Two "let rec" groups can be merged and type checking will still succeed
* An existing set of "let" bindings for functions can be made recursive and type checking will still succeed (because the order of construction of the let bindings mean that the types are necessarily independent)
* At the final binding of a recursive group, all bindings are considered "indpeendent". Thus the rule can be seen as a replacement for the existing rules for generalization, rather than a new rule.

The change will not break existing code, except that it may make some code more general and hence require a change in a signature file. One ramification of this rule is that members that get used generically within their own scope at non-regular types must be placed earlier in the declaration order. For example, this typechecks:

    let rec f x = x

    and g x = f 1

but the reverse order gives rise to different types (f is not generic)

    let rec g x = f 1

    and f x = x

While this is imperfect, it is relatively rare, since irregular recursive uses are relatively rare. It does introduce a new ordering dependency in F#, but

1. F# code is already left-to-right ordered
2. It is better than having to fully qualify the types for all members used generically, especially properties in generic classes

During a review of this for soundness with James and we found some interesting cases. Notably

* The process of determining the set of generalizable bindings must be a greatest fixed point
* The type variables for class definitions and member definitions should not act as a limit on the set of generalizable bindings. For example, given the following:

type Vec<'a>(x:'a,y:'a) =

  member v.x = x

  member v.y = y

  member v.Mul(w:Vec<'b>) : Vec<'a\*'b> = new Vec<'a\*'b>((v.x,w.x),(v.y,w.y))

The type of "X" does involve 'a, as does the type of Mul, but this is not an inference variable – it is a declared type inference variable.

#### Avoiding Quadtratic Cost in the Implementation

The first implementation of the above had a problem with quadratic cost. It turns out we can remove the quadratic behaviour as follows.

* During type checking we already keep a table of recursive uses of values, indexed by target value.
* This table is usually much smaller than the number of remaining forward declarations – e.g. in the pathological case you mentioned below this table is size 1.
* If a forward declaration does not have an entry in this table then its type can't involve any inference variables from the declarations we have already checked.
* So by scanning the domain of this table we can reduce the complexity down to something like O(n \* average-number-of-forward-calls).
* For a fully connected programs or programs where every forward declaration is subject to a forward call, this would be quadratic. However we do not expect callgraphs to be like this in practice
* The complexity is still dependent on the number of ungeneralized bindings. The complexity is something like:

O(n \* (average-number-of-forward-calls + average-number-ungeneralized-bindings)).

### Value Restriction Issues

This documents a corner case relating to false value restriction warnings for code such as that below

type Foo() =

    let mutable data : obj = null

    member this.Data1 = data

    member this.Data2

        with get () = data

        and set s = data <- s

Here the fact that we allow subsumption on assignment to fields means that “Data2.set”  is potentially generic – the input “s” could, in theory, be a value of any type. However, there is a .NET restriction that properties may not be generic. Hence “Data2.set” is not generalized over the type of “s”.

This is previously reported as a value restriction. Instead, we now do not apply the value restriction to any function or member bindings, and instead leave these type variables involved in these bindings as unsolved, and ultimately choose arbitrary types for them. This means “s” gets type “obj” in the above example.

This also affects code that is not fully generalized, e.g.

                let f<'a> (x,y) = (x,y)

Here the types for x and y are not generalized because the user has explicitly mentioned the generalized type variables. Again we previously reported a value restriction. Now, in F# 2.0, the inferred type is

                f<'a> : obj \* obj -> obj \* obj

### Using Type Names as Constructor Functions

FAQ:

Why does F# allow “C” to be used as a function name when it is a class constructor, even when C is generic?

Answer:

*For example, KeyValuePair(k,v) now becomes KeyValuePair<\_,\_>(k,v). And using the a single constructor in a generic type as a first-class value now becomes*

(fun (k,v) -> KeyValuePair<\_,\_>(k,v))

*which is a bit long – it almost feels like you should just be able to use “KeyValuePair” as a first class value if it’s not ambiguous.*

Note:

*Consider*

type Foo<'a>() =

    member x.f() = ()

type Foo<'a,'b>() =

    member x.f() = ()

let foo = Foo()  // Foo<'a,b> !!!

*We give an error here that says “Foo()” is ambiguous and tells user to select Foo<\_>() or Foo<\_,\_>().  We discussed the case in design meeting and decided to make it an error/warning/deprecation.*

### Auto-Open of Own Namespace

This feature is in the spec, but has not been widely disseminated.

An implicit open is performed on the namespace for the namespace fragment being defined. Thus

namespace A.B

does an auto-open on namespace A.B, and

module A.B.C

also does an auto-open on namespace A.B

### Implicit Resolution of Operators to Static Member Constraints

This feature is in the spec, but has not been widely disseminated.

Given an expression that consists of an identifier “OP” where

* OP is an infix operator
* name resolution doesn’t resolve OP to a value

then the expression elaborates to

(fun x y -> ((^a or ^b): (static member (OPNAME) : ^a \* ^b -> ^c) (x,y))

where x,y where x and y are fresh variables and  ^a, ^b and ^c are fresh type inference variables. For example, consider the following code:

type Foo(x : int) =

    member this.Val = x

    static member (-->) ((src : Foo), (target : Foo)) = new Foo(src.Val + target.Val)

    static member (-->) ((src : Foo), (target : int)) = new Foo(src.Val + target)

    static member (+) ((src : Foo), (target : Foo)) = new Foo(src.Val + target.Val)

    static member (+) ((src : Foo), (target : int)) = new Foo(src.Val + target)

let x = Foo(3) --> 4  // Succeeds

let y = Foo(3) --> Foo(5)  // Succeeds

let x2 = Foo(3) + 4  // Succeeds

let y2 = Foo(3) + Foo(4)  // Succeeds

### Warning 1182 (unused variables)

The options for F# 2.0 were

1. Make –warnon:1182 official with the design as it stands today (we’re definitely not going to discuss or make design changes here at this stage)
2. Leave it as it is and discuss whether we blog about it at a later date (there’s no urgency to blog about it is there?)
3. Take it out of fsc.exe, and only have it in fsc-proto.exe

We settled for (2).

One issue raised was around the conflict of the naming convention this feature uses with a common naming convention used by some F# developers (\_foo).  The concern is that we might not have the option to turn this feature on more officially in a future release because of it’s affect on these codebases.  However, it wouldn’t be a breaking change, and it’s almost impossible to plan ahead through the number of unknowns there would be in this discussion.

## Expressions

### Design Notes for Ranges over Enums

In F# 2.0 we disallow “..” ranges over enums.

type Suits =

  |  Diamonds  =  1

  |  Clubs  =  2

  |  Hearts  =  3

  |  Spades  =  4

let suitSequence = seq { Suits.Diamonds..Suits.Spades }

Although it would be a nice language feature, making it work involves a fair bit of work and testing (the underlying enum type can be several different types, and actually generating the IEnumerable of the correct type needs a bit of work)

We could consider the feature for F# vNext

## Object Model

### The Explicit Class Syntax

In a design discussion in April 2008, the question of removing the explicit class syntax altogether was discussed. The conclusion was: we should always be using the implicit construction syntax unless there is some technical reason not to. Now, we still do hit occasional things that force us to use explicit construction, though much less so than ever before. The complete list of the scenarios that force this is:

1. Classes that require multiple constructors, where each of those constructors calls a different base constructor.   Note this situation is rare.
2. Attributes and/or XMLDoc on the implicit constructor. There is currently no way to target attributes at the implicit constructor. This is easy enough to fix
3. CodeDom round tripping. This is a major problem: it will be very hard to make the CodeDom class generation invertible if it has to do anything complex to detect multiple constructors and choose or invent a “primary” one to be the implicit constructor.

### Required “this” and “static” in members

Allowing people to remove “this.” appears to add subtlety to the language, because it is evidently unclear if

member P = side-effect

is strict or delayed. Some people see the binding as “naturally strict” and others see it is “naturally delayed” – for example, we wouldn’t want removing an innocent and unused “this.” to change evaluation semantics. The original motivation for allowing the removal of “this.” was that users don’t like naming redundant, unused variables – yet removing that variable would also change evaluation.

Regardless of how we might resolve the evaluation semantics of the above binding, it seems that allowing the user to remove a harmless-looking “this.” does add an extra confusion-point to the language. F# is succinct enough, and there’s no need to seek further succinctness at the expense of making F# more subtle.

We can, however, expect to continue to get use requests to allow “this.” to be removed.

A similar FAQ about why “static” is required:

The static keyword is rather ugly. The reason it is needed and used in C#/C++ is because the method signature doesn’t explicitly specify the "this" argument and so you need to introduce a keyword to distinguish instance and static.

This response answers this well:

*Don’t like this – too subtle.  As it stands today, I* always *forget to put “this.” on members, and thankfully the compiler yells at me.  If instead this silently changed the meaning to “static” I think I (and many other programmers) would get very confused.  The suggestion has good syntactic elegance, but bad usability.*

### Design Notes for Interface Implementation Expressions

.NET has collection interfaces such as System.Collections.Generic.ICollection, IEnumerable etc. These can be annoying to implement by delegation. For example, implementing an ICollection<T> in terms of an ICollection<Option<T>> where the options are always Some(\_). We have to write this goop

interface IEnumerable<'key> with

      member s.GetEnumerator() = (keys |> Seq.map (fun v -> v.Value)).GetEnumerator()

interface System.Collections.IEnumerable with

      member s.GetEnumerator() = ((keys |> Seq.map (fun v -> v.Value)) :> System.Collections.IEnumerable).GetEnumerator()

With “interfaces by delegation to an expression” we could write just

interface IEnumerable<'key> = keys |> Seq.map (fun v -> v.Value)

This is, of course, just scratching the surface...

Jomo says:

*Would you be able to override the delegation so that interface implementations could be chained?*

*interface IFoo<'key> = contained.Foo() with*

*override x.FooMember() = contained.FooMember() - 1*

*This would be incredibly helpful in the scenario of unittesting gigantic interfaces (like IVs\*) Especially if you could default members to throw NotImplementedException for the root of the chain:*

*interface IFoo<'key> notimplemented with*

*override x.FooMember() = 0*

*Actually, now that I think of it, this latter part could be done with reflection as follows:*

*interface IFoo<'key> = NotImplemented<IFoo<’key>> with*

*override x.FooMember() = 0*

*NotImplemented would be a plain old function that takes IFoo<’key> and manufactures an implementation where each method throws an exception.*

Don says:

We should look at making this orthogonal with a delegation object expression:

               { <expr> with <overrides> }

That is spiritually similar to the record overriding we already have:

                { <expr> with <field-bindings> }

Hence

                let foo = { Foo() with override contained.FooMember() = 1 }

and

       type X()=

          interface IFoo<'key> = { Foo() with override contained.FooMember() = 1 }

Either way the self-referential call back to the outer self would look like this:

type X() as x =

     interface IFoo<'key> = { Foo() with override contained.FooMember() = x.FooMember() – 1 }

### Structural Types

An FAQ from Don Box:

*It sounds like records are nominally typed but tuples are structurally typed. I’m curious why OCaml took that position?*

Haskell, OCaml and F# all start with a core language that has no structural typing. They have a fairly simple, concrete core data language containing records, unions, tuples, etc. that is highly suited to type-inferred, immutable functional programming. Here data is concrete, efficient, predictable, simple-to-compile and simple-to-work-with.

When using this core language, the basic Haskell, F# and OCaml methodology (and indeed the typical approach in most real-world functional programming) is to “transform into” data into a strongly typed model (without any structural typing), transform the data, and then “transform out”. Very often this will involve transforming to/from untyped or partially typed schemas (XML, text files etc.). In F# we see people making some phases of this generic through the use of schema compilers/transformers (Expert F#, ch 9).

That said, OCaml certainly dabbles in structural typing – indeed the “O” in OCaml is the racing car of type-inferred structural typing systems. However the structural typing of OCaml is not used anywhere as much as you might think, and still only in routine programming rather than data modelling. Xavier Leroy once said “I think I used structural object types once somewhere in the compiler”.  Notably Haskell has not yet ventured in that direction. For both OCaml and Haskell the questions get very tied into the whole question of how OO programming should work in those languages.

For F# the obvious choice is to keep the core language (which we know works so effectively), and go nominal for the object model: we need a nominal object model for .NET interop in any case, and .NET is not a great compilation environment for structural types anyway.

That said, structural typing absolutely has its uses, most convincingly in modelling and query settings, as displayed by C# 3.0, and now M. As a result I can certainly see us adding structural typing (e.g. anonymous records) into F# at some point – indeed we almost added it for V2.0.

### Immediately Recursive Types

Question:

*Is there a reason this doesn’t work in F#:*

  type Tree = string \* list<Tree>

*I’ll observe that using records rather than tuples will work:*

  type Tree = { Operator : string; Children : list<Tree>; }

Answer:

*Types like these are useful in some situations.*

*The simplest reason for F# is that the first is type abbreviation, and needs to be erased during compilation by rewriting all occurrences to the right hand side. This erasure would not terminate.*

*However, types like the one shown are not allowed in OCaml either (unless you provide an explicit –rectypes command line option), and nor Haskell IIRC. This is due to type inference issues rather than representation issues - neither Haskell nor OCaml use a typed intermediate language, hence they would just represent the given type as a tuple. Most type inference systems will fail to terminate given these equations. However the –rectypes in OCaml address this problem through an approximation of some kind: I believe they only guarantee to expand these recursive equations 2 or 3 times when it’s not obvious the expansion is needed. This seems to be enough for all known user code.*

Follow up question:

*it sounds like an implementation issue more than a language/type system design issue?*

Follow up answer:

*People design different type systems for different reasons and engineering goals, e.g. checkability, expressitivity, interoperability, performance, inference. These days we can design a type system for more or less anything, with more or less any properties, up to tradeoffs. Termination of type checking or type inference is certainly one useful property, but it’s not necessarily the most critical one. Choosing between these tradeoffs is an inherent part of type system design.*

*However, I feel the construct below has problems beyond this. All the code I’ve ever seen that makes use of this has felt “very hard to understand”, sitting at some extreme end of functional programming.  Note that In F# you would just make the type nominal as follows:*

  type Tree = Tree of string \* list<Tree>

*which means you have to tag  consumption and creation sites:*

  let rec f (Tree(nodeName,children)) = [nodename] @ List.collect f children

*This makes type inference much easier than for the tagless version*

  let rec f (nodeName,children) = [nodename] @ List.collect f children

*which gives an “infinite type” error in F#. But perhaps more importantly I feel the second code is very hard to understand since it really gives you no point at which to start understanding the code – no tags, no types, no nothing. This is OK for some simple code like*

  let f (x,y,z) = (z,x,y)

*but I feel it is not OK for anything recursive.*

## Methods and Functions

### “mutable” in argument position

There is frequently a suggestion to allow “mutable” in argument position. This is how we write mutable classes, i.e.

type C(x:int, y:int) =

     let mutable x = x

     let mutable y = y

     member this.X = x

     member this.Y = y

rather than

type C(mutable x:int, mutable y:int) =

     member this.X = x

     member this.Y = y

Don says*: I find the latter visually confuses the API to the class with the implementation (the constructor signature is part of the API, the mutability part of the internal implementation). I also feel this is code-compactness-taken-too-far. And I really just dislike seeing the dreaded “mutable” in argument position.*

Brian says: *I do agree with Don’s point that putting “mutable” in the argument position is unfortunate (as it’s an implementation detail and so should not project into the textual part of the code that is the interface/signature description of the thing).*

## Structs

### “let” in structs

“let” bindings are not allowed in structs, neither for immutable nor mutable bindings. Here’s some commentary on this:

*the rule “let is not allowed in structs” is already creating a potentially confusing syntactic asymmetry between structs and classes – but it seems necessary and is at least easy to explain (including in an error message, which is where it will be communicated to a large percent of users).  Changing that to “let is not allowed in structs except when it is of the form ‘let mutable <foo> = <foo>’”  would make the asymmetry both harder to explain (cause there would be lets in structs), and is less intuitive.*

“let” should almost certainly be allowed for function bindings (which become members).

## Sequence and Computation Expressions

### History of Incomplete matches in “for” loop patterns

F# 1.9.3 and before used to let you use incomplete matches in sequence expressions, e.g.

seq {for Some r in [Some "a"; None; Some "b"] -> r}

F# 1.9.4 started to give a warning:

C:\fsharp2\staging\src\tests\fsharp\core\comprehensions\test.ml(289,41): warning FS0025: Incomplete pattern matches on this expression. The value 'None' will not be matched Sequence expressions nvolving incomplete matches on 'for' and 'let' constructs are deprecated and will raise an IncompleteMatchFailure exception in a future revision of the F# language. Please use a separate match expression instead, adding 'yield! []' for the failing branch.

We later removed the feature.

Note you can use “()” on branches that yield nothing

    [ for opt in [Some "a"; None; Some "b"] do

         match opt with

         | Some r -> yield r

         | None -> () ]

## Units of Measure

### Systematic Design Patterns and Overloading for Unit Conversion

Here is a design suggestion on a systematic use of overloading for unit conversions:

*Something type directed may well fit with F# (since we use statically resolved, type directed overloading for a bunch of things). e.g. this looks pretty convincing (it doesn’t compile right now, though I think there’s no reason it can’t/shouldn’t)*

[<Measure>]

type kg =

    /// This gives the name of the standard SI unit

    static member SIKind = "Mass"

    /// This gives the scaling factor to the standard SI unit

    static member ScalingToSI = 1.0

[<Measure>]

type g =

    /// This gives the name of the standard SI unit

    static member SIKind = "Mass"

    /// This gives the scaling factor to the standard SI unit

    static member ScalingToSI = 0.001

kind<kg>  // = "Mass"

kind<g>  // = "Mass"

let x1 = 127.0<kg> |> kg

let x2 = 127.0<g> |> kg

The necessary overloads would be something very close to this:

let inline kind< [<Measure>] ^u when ^u : (static member SIKind : string) > =

    (^u : (static member SIKind : string) ())

let inline kg (x:float< (^u) >) =

    (^u : (static member ScalingToSI : float) (x))

Andrew replied:

*Neat! But now I want to convert from a density in float<g/cm^3> to float<kg/m^3> (assuming base conversions g to kg and cm to m).*

Don replied: I think this would work

let inline massToSI (x:float< (^u) >) =

    float x \* (^u : (static member MassToSI : float) (x))

let inline distanceToSI (x:float< (^u) >) =

    float x \* (^u : (static member DistanceToSI : float) (x))

let inline densityToSI (x:float< ^mass / ^dist ^3 >) =

    float x \* massToSI 1.0 / cube (distanceToSI 1.0)

## Scripting

The design of scripts in F# 2.0 led to many interesting design discussions.

### Should Scripts be “Source Code Assemblies”?

Some design team members (e.g. Don) felt strongly that scripts should be “source code assemblies”, e.g. Could a script be strong-named-and-signed using identical identity logic to a .NET assembly?

Script reference:

#load @"http://code.msdn.microsoft.com/FinancialFunctions.fsx, **PublicKey=A06AC49443C035A9…"**

The script:

#strongname "<some-embedded-bytes>"

Where some build or web-delivery process embeds the bytes into the script on Luca's side? If he wanted to delay-sign he could use something like this:

#delaysign "<some-embedded-bytes>"

For the version, you would then have the usual .NET choice of embedding the version in the name or not, i.e. in this case the assembly name would be "FinancialFunctions":

#load @"[http://code.msdn.microsoft.com/FinancialFunctions.fsx@PublicKey=A06AC49443C035A9, Version=1.0.0.0](http://code.msdn.microsoft.com/version-1.2.0.0/FinancialFunctions.fsx@PublicKey=A06AC49443C035A9...)"

And in this case it would be"FinancialFunctions-v2"

#load @"[http://code.msdn.microsoft.com/FinancialFunctions.fsx@PublicKey=A06AC49443C035A9, Version=2.0.0.0](http://code.msdn.microsoft.com/version-1.2.0.0/FinancialFunctions.fsx@PublicKey=A06AC49443C035A9...)"

## F# Interactive

### F# Interactive and AssemblyResolve Event

FSI.EXE changes the behaviour of AssemblyResolve event, and in particular on an assembly load failure it

* prints an exception message
* throws a different exception which can’t be caught (except by a generic exception handler)

This is because fsi.exe adds to the AssemblyResolve event handler. This is to support #r on paths, e.g.

#r @"c:\foo\bar.dll”

Fsi.exe also adds a resolve handler at the end of the chain of assembly handlers that reports a good error message when an assembly load fails (a common case in fsi.exe when the assembly reference set is incomplete or reference paths have not been set up correctly). This means catching assembly load failures won’t work very well, because an error will already have been shown.

### Known Limitation: F# Interactive & IIS/WCF

There are some interesting and important scenarios where F# Interactive can’t be used in F# 2.0. For example, consider playing around randomly with WCF types and methods from FSI. This code:

#r @"System.ServiceModel.dll"

let factory1 = System.ServiceModel.Activation.ServiceHostFactory()

let service1 = factory1.CreateServiceHost("", [| System.Uri "<http://foo>" |])

gives:

System.InvalidOperationException: 'ServiceHostFactory.CreateServiceHost' cannot be invoked within the current hosting environment. This API requires that the calling application be hosted in IIS or WAS.

Which immediately raises the question “can I run my instance of FSI.EXE as if it is hosted in IIS”? Likewise this:

let factory2 = System.ServiceModel.Activation.WebScriptServiceHostFactory()

let service2 = factory2.CreateServiceHost("", [| System.Uri "<http://foo>" |])

gave the same error.

### Known Limitation: F# Interactive & Transportable Code

It is not possible to use interactively compiled code in other application domains or processes. For example, I was playing around with creating “remote” agents in other application domains, and this bit fails, because the F# dynamic assembly is created in one particular app domain and is not saved to disk as a DLL.

One possible direction would be to support a #save FSI directive here. This would save the current DLL being generated and start a new dynamic DLL. The current DLL (and any past, saved DLLs) could then, I think, be used from other app domains, and serialized and sent across the wire to other processes or a cluster.  Fixing this sort of thing could potentially be part of a V1.1/V2.0 focus on applied distributed computing with F#.

type Agent() =

    inherit System.MarshalByRefObject()

    let mbox = new MailboxProcessor<\_>(fun mbox ->

        let rec loop n =

            async {

                printfn "waiting..."

                let! msg = mbox.Receive()

                printfn "msg = %A, n = %d" msg n

                match msg with

                | Choice1Of2 i ->

                    return! loop (n+i)

                | Choice2Of2 (reply: AsyncReplyChannel<int>) ->

                    reply.Reply n

                    return! loop n

            }

        loop 0)

    do mbox.Start()

    member x.Send n = mbox.Post (Choice1Of2 n)

    member x.AsyncGet n = mbox.PostAndAsyncReply (fun reply -> Choice2Of2 reply)

// note: adding a #save directive may be enough to make the generated code accessible across app domains

// #save

let agent = Agent()

let remoteDomain = System.AppDomain.CreateDomain("other domain")

System.Reflection.Assembly.Load("FSI-ASSEMBLY") // succeeds

let obj = remoteDomain.CreateInstanceFromAndUnwrap("FSI-ASSEMBLY", typeof<Agent>.Name)

# FSharp.Core Design Notes & FAQs

## Basic Operators

### “enum” only works on 32-bit

In F# 2.0, the 'enum' function only converts 32-bit enums (the common case). This was for two reasons, first the signature of the function is simpler

val enum : int32 -> 'e when 'e : enum<int32>

c.f.

val enum : 'u -> 'e when 'e : enum<'u>

Here the user has no idea what this "u" thing is all about. Second, it allows

enum<EnumType>(3)

which is neat.

EnumOfValue was added in F# 3.0 to allow enum literals for other types.

### Why both “\*\*” and “pown”

Question: why do we support both of these operators?

*I’m just not yet convinced that the user cares about the difference between the two operators below (their internal implementation details).  They just want to be able to take the power for any reasonable set of arguments.  I don’t know of any other language/math tool which differentiates these two – none of* [*C#*](http://msdn2.microsoft.com/en-us/library/system.math.pow.aspx)*,* [*matlab*](http://www.mathworks.com/access/helpdesk/help/techdoc/index.html?/access/helpdesk/help/techdoc/ref/pow2.html&http://www.mathworks.com/support/functions/alpha_list.html?sec=6)*,* [*python*](http://docs.python.org/lib/module-math.html)*,*[*mathematica*](http://reference.wolfram.com/mathematica/ref/Power.html) *or* [*C++*](http://www.cplusplus.com/reference/clibrary/cmath/pow.html) *makes this distinction.  The reason we might be different than these other tools is that our type system tend to push us to making some tradeoffs for these kinds of operators.  In particular, we want type-inference to flow as strongly as possible between the arguments of our operators – which requires we have fairly strict type signatures.   My assumption is that this was what motivated Don’s original proposal.  But I doubt we have any compelling reason to be different than these tools because of actual user functionality needs. Aside:  Interestingly, matlab does have a separate “pow2” function (derived from C++ “scalbn” and “ldexp”), which raises 2 to the given power.  Of course, for us this is just (1 <<< x) and (1 >>> x).  It also has a two argument version pow2(x,y) which returns x\*(2\*\*y).  But I don’t expect that is critical for F#.*

In F# 2.0 we decided to keep both. Some rationale in the email below:

*Don says: Here’s a C++ reference which basically says “there are two C++ versions of ‘pow’, both with the same semantics but one sucks and the other is actually fast when used with integer exponents”*

[*https://twiki.cern.ch/twiki/bin/view/Atlas/CalculatingIntegerPowers*](https://twiki.cern.ch/twiki/bin/view/Atlas/CalculatingIntegerPowers)

*“Intuitive performance expectations” is one reason why I don’t mind separating the two. Users will intuitively expect a certain performance profile from “pown  x n”. Furthermore we should (perhaps over time) find it fairly simple to reach that performance profile without rat-holing into a mess of floating point complexities. For example, we can just pattern-match up to n = N for some fixed N and call specialized code x\*x, x\*x\*x etc. This means we can leave the performance spec as*

*\*\*  = “call System.Math.Pow and you get whatever performance .NET gives you for that”*

*pown = “do the obvious fairly simple optimizations for different input types/low-indices that you can find in the published F# library source code otherwise use a multiplication loop”*

### Dynamic Evaluation of Basic Operators

Consider the evaluation of F# quotations via LINQ.

#r @"FSharp.PowerPack.dll"

#r @"System.Core.dll"

#r @"FSharp.PowerPack.Linq.dll"

open Microsoft.FSharp.Quotations

open Microsoft.FSharp.Quotations.Linq

    let Eval (q: Expr<\_>) = q.Eval()

    let f () = ()

    Eval <@ f ()  @> )

    Eval <@ let f (x:int) (y:int) = x + y in f 1 2  @>)

    Eval <@ let rec fib x = if x <= 2 then 1 else fib (x-1) + fib (x-2) in fib 36 @>

    Eval <@ if true then 1 else 0  @>

    Eval <@ 0x001 &&& 0x100 @>)

    Eval <@ 0x001 ||| 0x100 @>)

    Eval <@ 0x011 ^^^ 0x110 @>)

    Eval <@ ~~~0x011 @>)

    Eval <@ System.DateTime.Now @>

ASIDE: the explicit performance goal for this work is NOT to achieve the same performance as compiled, optimized F# code. Rather we are happy if we’re within a factor of 10x.

This runs into problems if we’re calling inlined functions, since LINQ tries to dynamically invoke these functions. Among other things, this checkin made some progress on allowing dynamic invocations (and thus quotation-evaluation) for inlined functions. In particular, we now emit method bodies for inlined functions. However

* This is difficult for functions that invoke member constraints. At some point we should move to have these functions accept explicit MethodInfo witnesses as arguments that give the solution to the member constraint. For now, member constraint invocations raise an exception if dynamically invoked. This means there are a bunch of functions in our library that can’t yet be dynamically invoked (see bug 3239)
* There are lots of cases where we can make this work correctly, and we’ve implemented dynamic invocation for (+), GenericZero, DivideByInt and GenericOne when instantiated at basic types. This means that our inlined Seq.sum, Seq.average work correctly
* Emitting member bodies for inlined functions would cause a problem when inlining is used to hide unverifiable code. As a result added one new attribute is added to mark methods where we really, really don’t want the code emitted:

    [<System.AttributeUsage (System.AttributeTargets.All,AllowMultiple=false)>]

    [<Sealed>]

    type NoDynamicInvocationAttribute =

        inherit System.Attribute

        new : unit -> NoDynamicInvocationAttribute

## List, Array, Option, Seq

### Should List.sum etc. be checked or unchecked?

This is an FSharp.Core.dll library design issue:

*Because of the way checked is implemented in F#, can’t we actually have our operators throw an exception only in checked? We just have different versions of the operators and auto-open the checked version.*

Decision:

*It’s hard for us to go down this route. For example, it’s hard for us to get*

*open Microsoft.FSharp.Operators.Checked*

*to change the meaning of List.sum.*

## Microsoft.FSharp.Reflection

### Why do we need “BindingFlags”?

We need to take BindingFlags into account in the reflection API. The simple fact is that in F# record and union representations can be private, even if the type itself is public. I’ve come to realize that we should be using BindingFlags.Public and BindingFlags.NonPublic to account for this. This affects the style of the reflection API, and is driving some changes.   In future releases of F# it’s possible individual record fields could be made private.

The guiding principle is simple:

* **If the user doesn’t specify BindingFlags.NonPublic then they won’t discover private record and union representations**
* **If the user specifies BindingFlags.NonPublic then they will be able to see and access private record and union representations**

We looked at four different API designs. In the end we chose to implement with BindingFlags optional and BindingFlags.Public the default.

## Microsoft.FSharp.Control

### F# Async Design Overview and Principles

For F# V1, the overriding aim of “async” is to provide a simple, language-integrated way of writing program fragments that do not block .NET/OS threads at synchronization points, which run efficiently as fully compiled code, and which are controlled in their introduction and use of multi-OS-thread concurrency. The mechanism is characterized by the presence of “async { ... }” expressions in user code.

Except in simple cases, the mechanism is a general workhorse that is embedded in some larger reactive/concurrent/parallel design pattern or architecture. Some sample intended applications of the mechanism are:

* Introductory asynchronous and parallel samples
* Fork-join
* Background workers (e.g. a spell checker)
* CPU-bound parallel computations (e.g. a scientific algorithm)
* Computational applications making use of asynchronous IO (e.g. a web crawler/aggregator)
* GUI components making use of asynchronous I/O (e.g. a silverlight component)
* Active objects and actors (e.g. any in-process Erlang algorithm)
* Other combinators and “design patterns” that introduce reactive agents or manage concurrent and parallel execution

The aim is to make a mechanism where the defaults are to “do the right thing” and to make reasoning about your program relatively simple in common user cases. At a high level, the async mechanism gives the user:

* **Language supported continuation management.** The process of taking the continuation of an operation, waiting for an event and invoking the continuation when the operation returns is implicit at let! and other binding points within the program.Try/finally and other error recovery mechanisms can be used in async program. It is the responsibility of the overall design pattern in which the construct is embedded to ensure cancellation conditions are set.
* **Simple introduction of explicit parallelism.** You must introduce parallelism explicitly using Async.Parallel, Async.Start, Async.StartChild, MailboxProcessor, etc.
* **A foundation for writing components that encapsulate common parallel, reactive and asynchronous design patterns.** e.g. MailboxProcessor
* **Controlled introduction of multiple OS threads.** You can reason about which OS thread your code executes on, and API operations that introduce additional OS threads or change execution to the .NET thread pool can be easily identified. The API makes it simple to write async I/O programs where no user code executes in the .NET thread pool.
* **Integration and interoperability with .NET 4.0 computation tasks.** Async values can be executed as .NET 4.0 tasks, and can synchronize with the completion of these tasks.

Terminological clarifications:

**Async**. Wherever we use the word “Async” we mean “potentially asynchronous at the level of OS threads”. We need to be very consistent about that

**Synchronization Context.** When used in official documentation this means System.Threading.SynchronizationContext.Current. In our own discussions we can use it informally to mean the distinction between “GUI thread”, “background thread”, “thread pool”, “GPU”, “cloud”, ...

**Multi-threaded, shared-memory, preemptive concurrent execution**  (also “Background concurrency” or “Nasty concurrency”)

== any execution that involves multiple OS threads executing user code which may access mutable shared memory or other non-concurrency-safe objects concurrently

== specifically .NET/OS thread pool or new .NET/OS thread running user code that is not the initial thread of an application.

**Event**. A GUI-thread event, including Winforms events, other .NET event declaration (all of which are GUI events), F# first class event values (when used in a way that follows the discipline that event handlers are triggered on the GUI thread), WPF events “revealed” as a .NET event declaration

We assume that users of the async { .. } mechanism will

* Be able to take a sample of one of the above design patterns and adjust it for their needs
* Understand the basic principles of an asynchronous I/O request and a callback
* Understand what an OS thread is
* Understand that an async program fragment is not the same as an OS thread
* Understand that the aim of “async” is to not block OS threads
* Understand that a typical concurrent or reactive design pattern uses OS threads in various ways
* Understand the dangers of using shared memory concurrency and try to avoid it except where needed
* Understand that some operations in a .NET API may block the current OS thread and that these should be avoided in async blocks
* When programming a GUI or Silverlight
  + understand the need to keep response computations relatively short
  + understand the need to avoid the use of background concurrency except where needed
  + if they use nasty concurrency, understand that GUI objects are not safe for concurrent access, and are owned by the GUI thread, and must only be accessed by code running on the GUI thread
  + understand the principle of single-threaded ownership of other mutable objects

We assume that experts who write new combinators or design patterns will

* Have a very good understanding of all of the above topics
* Understand the notion of a “synchronization context”
* Understand what operations in .NET APIs block the current thread
* Understand design patterns such as APM
* Understand .NET GUI events

### Async “Stay In Context”

**Async Operations Stay In Their Synchronization Context By Default (unless something explicit tells you otherwise)**. That is, unless explicitly documented otherwise, all API operations (e.g. FromBeginEnd, StartChild, AsyncRead, Async.Parallel) run **constituent user computations** and any **continuation** via the sync context active when the execution of the async computation starts. This means that when these operations are started from a GUI thread, then any constituent user computations will also be run on the GUI thread. If there is no sync context then the .NET thread pool is used.

The only ways to get to another sync context are:

* explicit use of a background component such as MailboxProcessor or BackgroundWorker
* explicit use of operations taking a startInThreadPool =true
* explicit API calls such as SwitchTo\*
* explicit use of RunSynchronously, which has an implied SwitchToThreadPool

By implication, if you do not use any of the above, then no multi-OS-threaded concurrency is introduced to a GUI or Silverlight application. Note that in some contexts, such as Midori, no multi-OS-threaded concurrency is permitted at all.

When Async.RunSynchronously is used from a thread with a synchronization context (e.g. from F# Interactive, or from a GUI application where a button deliberately starts a blocking computation), an immediate SwitchToThreadPool is executed to ensure the GUI thread is not blocked.

Note: Since the initial design discussions on this topic, I’ve been reviewing various bits of async code for GUI-thread correctness, and overall I am genuinely surprised how hard it is to reason about the current OS thread of an Async using our current API: very innocuous asyncs such as AsyncPostAndReply run their continuation in the thread pool even when started on the GUI thread. This is just very dangerous. We really have to either ban the use of “async” on threads with a sync context, or adopt the above rule. So I do think this design change is very important.

### Trampolines

Since async code is essentially an inversion-of-control framework, we rely a lot on tail calls to our continuations. Unfortunately .NET 2.0 x64 JIT has a bug that sometimes make the compiler emit the regular call instead of the tail call, and that makes our async stuff stack-dive. So our async library implements a simple trampolining scheme to circumvent the issue.

We add two classes:

    [<AllowNullLiteral>]

    type Trampoline() =

        let mutable cont : (unit -> unit) option = None

        let mutable bindCount = 0

        member this.ExecuteAction (firstAction : unit -> unit) =

              ... // execute action and then cont while cont is not None

        member this.IncrementBindCount() = …

        member this.Set action = cont <- action

    type TrampolineHolder() =

        let mutable trampoline = null

        member this.Protect firstAction =

            trampoline <- new Trampoline()

            trampoline.ExecuteAction(firstAction)

        member this.Trampoline = trampoline

And AsyncParams have a trampoline holder:

    type AsyncParams<’T> ={ ...; currentTrampolineHolder : TrampolineHolder }

Bind installs the continuation to a current trampoline in trampoline holder:

                    let trampoline = args.currentTrampolineHolder.Trampoline

                    if trampoline.IncrementBindCount() then

                        trampoline.Set(fun () -> invokeA p1 args |> unfake); FakeUnit

                    else invokeA p1 args

Now whenever we switch threads/escape the trampoline that is on stack (Async.SwitchToThreadPool, Async.SwitchToSyncContext and all others), we call args.currentTrampolineHolder.Protect, and this creates a new trampoline that will protect the continuation executing on this new thread:

// This should be the only call to QueueUserWorkItem in this library. We must always install a trampoline.

let queueWorkItemWithTrampoline (currentTrampolineHolder:TrampolineHolder) f =

    if not (ThreadPool.QueueUserWorkItem(fun \_ -> currentTrampolineHolder.Protect f)) then

       failwith "failed to queue user work item"

let switchToThreadPool() : Async<unit> =

    protectedPrimitive(fun args ->

        queueWorkItemWithTrampoline args.currentTrampolineHolder (fun () -> args.cont () |> unfake) |> fake)

### Bind hijacking check

Bind hijacking check is there to prevent the stack growing unbounded in case JIT does not respect tail calls. This may happen in:

* While or for loops inside an async { ... }
* Recursive async definitions of form “let rec a = async { .... a ... }

Current implementation of asyncs has hijack checks in async.Bind, async.For, async.While and async.Delay primitives. This covers any asyncs that can be created with async { ... } syntax. It is still possible to create asyncs that stack-dive using raw primitives, e.g.:

let rec a : Async<unit> = async.Using((null : System.IDisposable), fun \_ -> a)

### Splitting up of AsyncParams

I have played a bit more with this stuff, and I guess I have found the best solution 

Remember that I am attacking this inefficiency in bind:

let bindA p1 f  =

    unprotectedPrimitive (fun args ->

        if args.token.IsCancellationRequested then

            ...

        else

            let args =

                let cont a = protect args.econt f a (fun p2 -> invokeA p2 args)

                { cont=cont;

                  ccont=args.ccont;

                  econt=args.econt;

                  blocked=args.blocked;

                  token=args.token

                  currentTrampolineHolder = args.currentTrampolineHolder

                }

            ...

Bind recreates args structure on every invocation although it only changes one field of it.

In  the previous episodes we:

1. Split the args structure into cont parameter and “rest of args” parameter:

let bindA p1 f  =

    unprotectedPrimitive (fun cont args ->

        if args.token.IsCancellationRequested then

            ...

        else

            let cont a = protect args.econt f a (fun p2 -> invokeA p2 cont args)

...

Gave us much less GCs but slower performance due to extra parameter passing.

1. Made the cont field of AsyncParams mutable. Gave us less GCs, faster performance but we have lost static typing.

It occurred to me that it is possible to go the third way – do the restructuring suggested in 1) but keep cont a field of a record:

    [<NoEquality; NoComparison>]

    [<AutoSerializable(false)>]

    type AsyncParamsAux =

        { token : CancellationToken;

          econt : econt;

          ccont : ccont;

          blocked : System.Threading.Thread;

          currentTrampolineHolder : TrampolineHolder

        }

    [<NoEquality; NoComparison>]

    [<AutoSerializable(false)>]

    type AsyncParams<'T> =

        { cont : cont<'T>

          aux : AsyncParamsAux

        }

Now bind only reallocates the smaller AsyncParams record, keeping the “mostly immutable” part of it intact:

        let bindA p1 f  =

            unprotectedPrimitive (fun ({ aux = aux } as args) ->

                if aux.token.IsCancellationRequested then

                    cancelT args

                else

                    let args =

                        let cont a = protect args.aux.econt f a (fun p2 -> invokeA p2 args)

                        { cont=cont;

                          aux = aux

                        }

                                                …

This works beautifully. Here are my results:

c:\work\asyncPerf>runTest baseline

Real: 00:00:28.557, CPU: 00:00:27.783, GC gen0: 16801, gen1: 2, gen2: 0

c:\work\asyncPerf>runTest newTrampolines

Real: 00:00:22.604, CPU: 00:00:22.386, GC gen0: 17565, gen1: 2, gen2: 0

c:\work\asyncPerf>runTest newTrampolinesMutableParams

Real: 00:00:25.111, CPU: 00:00:24.882, GC gen0: 12224, gen1: 2, gen2: 0

c:\work\asyncPerf>runTest newTrampolinesSplitInMem2

Real: 00:00:21.426, CPU: 00:00:21.278, GC gen0: 14514, gen1: 2, gen2: 0

This new approach:

1. Keeps static typing in async implementation
2. Lowers GC pressure significantly compared either with current sources or with trampolines
3. Gives better raw performance than “mutable AsyncParams” version (although higher GC pressure).

I suspect two reasons for c). Primo, there are up- and down-casts going on in mutable async version. Secundo, with mutable AsyncParams, args record becomes an extremely long-lived gen2 object, and we are constantly updating its field with a reference to gen0 object (continuation closure). This is an expensive operation (GC needs to mark the card for inter-gen pointer on every assignment and then scan the card on every gen0 GC).

## Microsoft.FSharp.Linq

### Query design

General library support for queries is provided by query operator:

/// Evaluate the quotation expression by first converting to a LINQ expression tree

/// making use of IQueryable operators and then executing expression tree

///

/// Exceptions: <c>InvalidArgumentException</c> will be raised if the input expression is

/// not in the subset that can be converted to a LINQ expression tree

val query : Quotations.Expr<'T> -> 'T

Typical use is this:

query <@ seq { for c in db.Employees do if c.EmployeeID < 0 then yield c } @>

The quotation passed to query operator is in terms of sequences of (queryable) data. Query operator first translates that quotation into another quotation that uses LINQ-pattern methods, so that the above will become *something like*:

<@ db.Employees.Where(fun c -> c.EmployeeID < 0).Select(fun e -> e) @>

Then the resulting quotation is translated into expression tree and compiled.

### Equality and comparison in queries

When F# quotation is translated into expression tree, equality/comparison operators are translated according to F# rules, and <@ foo < bar @> becomes

Expression.LessThan(<foo>, <bar>, false, <*minfo for F# equality operator*>)

However Linq providers refuse to translate it into e.g. SQL less-than operator. Therefore two different quoatation-to-expression-tree translation modes are implemeneted. *Design issue:* *will it be possible to reconcile those two translation modes?*

Also, comparisons with System.Nullable<> types are not implemeneted. The above code would fail to compile if EmployeeID was a Nullable<int>. Current workaround is to write

fun c -> c.EmployeeID.HasValue && c.EmployeeID.Value < 0

# Library-only Language Features

## Static Optimizations

The F# compiler implements a “static optimization” when **--compiling-fslib** is enbled. This is used in FSharp.Core.dll. The grammar is:

<expr> := <expr> <static-optimization> and ... and <static-optimization>

<static-optimization> := when <static-condition> = <expr>

<static-condition> :=

<typar> : <type>

<typar> struct

true

Static optimizations are alternative versions of an expression. The alternative version is used whenever the expression is inlined and the static condition resolves to true. The alternative version must have **identical** semantics to the “main” version, and hence the feature is **very dangerous** to use.

The three conditions resolve as follows:

* <typar> : <type> is true if, after instantiation, the **head types** are equivalent (i.e. List<int> = List<string>) . It is not, despite appearances, equivalence of **types**. In practice the <type> is **always** a non-generic type.
* <typar> : <typar> is used to represent a condition which is “false” for non-inlined code and “true” for inlined code. The allows FSharp.Core to bake slow, reflective implementations of some functions (e.g. addition).
* <typar> struct is true if, after instantiation, the type parameter is a struct type.

It is common to get user requests to enable this as a first class language feature. Here is a typical response:

*While we understand the power of this feature, we feel it is not yet ready for prime-time in the productized version of F#. The history here is that it was added to the library for a very specific purpose – expressing a series of code generation optimizations that we would otherwise have had to add to the F# compiler itself.  Having the optimizations expressed in library source allowed us to avoid that complexity. However the feature suffers from numerous edge conditions and should, in principle, be tied to some kind of static typing discipline, rather than adding an additional form of adhoc overloading to the language.*

*If there is code that you absolutely can't make performant without this feature, after taking into account some reasonable code changes, then please do let us know.*

## Mutating tail-cons of lists

The F# compiler implements a library only construct that allows you to mutate the tail cons of a list:

let setFreshConsTail cons t =

cons.(::).1 <- t

This is used to mutate the tail cons of “fresh”, i.e. unpublished, unaliased cons cells, e.g.

let rec appendToFreshConsTail cons xs =

match xs with

| [] -> cons

| h::t ->

let cons2 = [h]

setFreshConsTail cons cons2

appendToFreshConsTail cons2 t

## Embedded IL

The F# compiler implements two “embedded IL” constructs when **--compiling-fslib** is enbled. This is used in FSharp.Core.dll. These enables:

* Embedded instructions
* Embedded types

Historically the number of constructs accepted as embedded IL was very large (e.g. it included the full range of method calls, field accesses, branching instructions etc. that make up Common IL). However in practice only a much smaller and simpler subset of embedded IL constructs are used in FSharp.Core.dll.

Uses of embedded IL such as (# "conv.i4" #) leak through to the external specification of F# because these end up appearing in the binary optimization metadata of F# assemblies such as FSharp.Core.dll. Hence it is important to at audit the use of these constructs, specify their binary representations and eliminate old code related to processing unused portions of the formats. Also, prior to V1, it is useful to seek to reduce the use of embedded IL.

Notes:

* Embedded IL includes some “fake” IL instructions representing operations on multi-dimensional arrays, among other things.
* A small number of additional embedded IL constructs are generated through the process of type checking F# code. These may appear in the F# binary formats. These include
  + **ldfld**/**stfld**/**ldflda**/**ldsfld**/**ldsflda**, used in the elaborated form of accesses to IL fields
  + **ldvirtftn**, used as a way to implement a “non-null” check that triggers a NullReferenceException is an object references is null
  + **nop**, used as an F#-compiler internal marker

28/9/2009: Here is a dump of the embedded IL we use in FSharp.Core

TYPES:

!0[]

!0[0 ...,0 ...]

!0[0 ...,0 ...,0 ...]

!0[0 ...,0 ...,0 ...,0 ...]

!0&

native int

!0\*

INSTRUCTIONS:

ldlen.multi, ldelem.multi, stelem.multi, newarr.multi, ldelema.multi

ldlen, ldelem.any, stelem.any, newarr, ldelema

isinst, castclass, box, unbox.any, sizeof, ldnull

ldobj, stobj, sizeof, localloc

ceq, clt, cgt, clt.un, cgt.un

add, sub, div, mul, add.ovf, add.ovf.un, sub.ovf, sub.ovf.un, mul.ovf, mul.ovf.un, div.un, rem, rem.un, neg

and, or, shl, xor, shr, shr.un, not

throw, rethrow

conv.i1, conv.i2, conv.i4, conv.i8, conv.i, conv.u1, conv.u2, conv.u4, conv.u8, conv.u, conv.r4, conv.r8, conv.r.un, conv.ovf.i1, conv.ovf.u1, conv.ovf.u2, conv.ovf.i2, conv.ovf.i, conv.ovf.u, conv.ovf.u1.un, conv.ovf.u2.un, conv.ovf.u8, conv.ovf.u8.un, conv.ovf.i8, conv.ovf.i8.un, conv.ovf.u, conv.ovf.u.un, conv.ovf.i, conv.ovf.i.un

ldtoken

ilzero

call "call instance int32 [mscorlib]System.String:: GetHashCode()"

# Binary Formats (Notes)

F# uses 4 binary formats in its compiled DLLs

* The F# signature resource
* The F# optimization resource
* The F# quotation resource for ReflectedDefinition literals
* The F# quotation byte arrays for quotation literals

# F# and Serialization

The .NET Framework includes at least 5 serializers and about 10 serialization “programming models” (i.e. ways of declaring what data should be serialized and what results). This document describes the current status of the interaction of these technologies with F#.

The problems we have are, in priority order:

1. Union types require a funky KnownTypes declaration to enable serialization. Fundamentally nothing we can do about this. Examples shown further below.
2. DataContracts can’t be applied to union values because we can’t apply attributes to union fields. This is actually easy to fix and could go in an SP.
3. Immutable values are not serializable in partial trust, e.g. F# tuples, list, option, set, map, union values. Nothing we can do about this without CLR changes
4. We need clear user guidance and documentation from the F# perspective – it’s very easy to get lost in this space. But if you stick to a charted path, the results can be quite pleasing. The walkthrough below is a good first step.

### Requirement: Relevant Serializers

The most relevant serializers are:

System.Runtime.Serialization.Json.DataContractJsonSerialization

System.Runtime.Serialization.DataContractSerialization (XML)

System.Runtime.Serialization.Formatters.Binary.BinaryFormatter

Two less relevant serializers are:

System.Runtime.Serialization.Formatters.Soap.SoapFormatter

System.Xml.Serialization.XmlSerializer

Note: These serializers can be used directly from F#, many samples are given in the rest of this document. For testing, work directly with the serializers rather than working with some upstream technology.

### Requirement: Stability of serialization formats

Serialization formats for F# types or any types interacting with F# programming must be stable and documented.

### Requirement: Interaction with Partial Trust and Security Transparency

Serialization should, where possible, be “neutral” w.r.t. partial trust

* more things may be enabled and start working with full trust
* serialization formats must not change between full and partial trust
* where possible, the overall F# programming model should not be influenced by partial/full trust decisions. However, this is problematic for serialization, where strong restrictions do apply in partial trust.

Some ways of declaring types to be serializable are only available to non-transparent code, e.g. implementing ISerializable. FSharp.Core.dll is a transparent assembly.

### Requirement: Stability across platforms

Serialization should be “stable” across platforms, e.g. .NET CF and Silverlight. That is, the modes of access, the wire formats etc. must be stable, and we must know the state of seerialization support on each platform.

## Overview of Serializers

### System.Runtime.Serialization.Formatters.Binary.BinaryFormatter

This uses the “all fields” programming model:

* Requires “Serializable” bit in the IL type, compilers set this
  + C# does this when seeing the [Serializable] attribute
  + F# does this all the time
* Defaults to serialize all the fields, public and private
* Can mark fields as NonSerializable
* Customize serialization with ISerializable, but can’t do that in a transparent assembly
* Customize serialization with serialization events (OnSerialization, OnDeserialization, ...)

The output of BinaryFormatter is “tightly coupled”

* Writes the type name, assembly name, assembly version
* Needs the same assembly on deserialization. This is a bad requirement that leads to extreme fragility of serialized data.

BinaryFormatter is highly dependent on desktop CLR

* not on Silverlight
* not on CF
* if on Mono, certainly not the same serialization format.

### System. Runtime.Serialization.DataContractSerialization and System. Runtime.Serialization.Json.DataContractJsonSerializer

DataContractSerializer is a contract-based serializer which writes XML. The JSON serializer is functionally very similar to DataContractSerializer, except it writes JSON instead of XML

Charcteristics of a contract-based serializer:

* The user provide a System.Type which is the root type of the object graph being serialized and deserialized
* Uses a very information-minimal format on the wire. Most information is left implicit. The same formatted data can match many types.
* Each type can declare a set of “known types” (KnownTypeAttribute)
* It will not write assembly name for known types (?)
* It will write type names for unknown types.
* Known types can also be specified when constructing the serializer/deserializer
* Known types can be declared on classes. These apply during the processing of the subgraph for that class

This serializer does not require a public default constructor, and there is no way to tell the serializer to use the default constructor. Instead the following “back door” is used to construct objects as uninitialized:

* System.Runtime.Serialization.FormatterServices.GetUninitializedObject. Other the other “back door” hooks in that class may also be used in the implementation – this is not entirely clear, e.g. System.Runtime.Serialization.FormatterServices.GetSafeUninitializedObject

However in patial trust the serializer does require a public default constructor.

### System.Runtime.Serialization.Formatters.Soap.SoapFormatter

SoapFormatter is marked obsolete . It uses a similar model as BinaryFormatter

* Will hit it in “old remoting scenarios”
* Even for .NET remoting we recommend to use binary formatter
* Obsolete message tells you to use binary formatter
* not on Silverlight
* not on CF (???)

### System.Xml.Serialization.XmlSerializer

Note: this serializer is harder to use from F#, because of the requirement for a parameterless constructor.

This is a contract based serializer.This means

* Will not write assembly name or type name for known types.
* Will write type names for unknown types.
* Each type can declare a set of “known types” (XmlIncludeAttribute)
* Known types can also be specified when constructing the serialzier/deserializer
* Known types can be declared on classes. These apply during the processing of the subgraph for that class

* must provide a System.Type which is the root type of the object graph being serialized and deserialized
* will only serialize public fields and properties of public types
* Deliberate schema compat between
  + List<T> and T[] and everything else that looks like IEnumerable
  + Dictionaries will all “look alike”
* requires a public default constructor, no way to tell the serializer to use the default constructor
* requires setters on properties
* to customize, use XmlElementAttribute, XmlTypeAttribute, .. These are designed to control exactly how things appear on the wire.
* Comes with tools for XML/Object mapping, i.e.
  + xsd.exe (thin wrapper over System.Xml.Serialization.XmlSchemaImporter, System.Xml.Serialization.XmlSchemaExporter)
* Can customize IXmlSerializable, lets you “write whatever you want to an XmlReader, you’re responsible for reading it back”
* Does not preserve the object graph at all
* Throws exception when cycles detected (won’t stack overflow)
* Supports serialization events (OnSerialization, OnDeserialization, ...)
* (is on Silverlight)
* (on CF 2.0??? 3.5???)

This serializer is used for the following upstream technologies:

* Used for asmx
* Used for a number of WCF scenarios
* More general than DataContractSerializer, can produce the full range of Xml

## Overview of Serialization Programming Models

F# users who need to do serialization using a .NET Framework serialzier must consider how to fit their types to an existing “serialization programming model”. This means “what do I have to write to make my values serializable, and what effect will it have”.

Be aware that F# users may be coming to serialization after reading a C# code sample (e.g. WCF or Silverlight), a blog, an existing F# code sample or a book.

The following are the most relevant programming models:

* **“All field serialization”**

At the IL level this is indicated by a bit in the IL metadata for a type in an assembly. System.Reflection makes this look like a regular custom attribute.

In C# this is indicated by the use of the [<Serializable>] attribute on a type. This attribute is extremely unfortunately-named.

In F# this is the default for all type. It can be suppressed using [<AutoSerializable(false)>]. In F# the [<Sytem.Serializable] attribute is applied to all types by default. The upside to this is that, with the exception of union types, closures, delegates and object expressions, F# types can generally be automatically used (in full trust) with either BinaryFormatter or DataContractSerializer. The downside is that the F# user must explicitly remove the attribute.

* “POCO serialization” (Plain Old C# Objects)

In C# this is indicated by a type without any attributes, usually with settable fields or properties.

In F# this is the default for types with [<AutoSerializable(false)>].

* “DataContract serialization”

In F# and C# this is indicated by the use of the [<System.Runtime.Serialization.DataContract >] attribute on a type. Adding the [<System.Runtime.Serialization.DataContract >] and [<System.Runtime.Serialization.DataMember >] attributes, with various flags and tweaks such as a set of “known types” or Name="Xyz" for DataMember. In this case the user is explicitly attempting to annotate the type, e.g. with a WCF data contract.

Somewhat unbelievably, there are at least 5 other “serialization programming models” in the .NET Framework, to quote:

[DataContractSerializer...] supports more than just the types marked wth DataContract attribute. It supports serialization of the following kinds of types in the default mode.

1. CLR [built-in types](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconthenetframeworkclasslibrary.asp)
2. Byte array, [DateTime](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemdatetimeclasstopic.asp), [TimeSpan](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemtimespanclasstopic.asp), [GUID](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemguidclasstopic.asp), [Uri](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemuriclasstopic.asp), [XmlQualifiedName](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemxmlxmlqualifiednameclasstopic.asp), [XmlElement](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemxmlxmlelementclasstopic.asp) and [XmlNode](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemxmlxmlnodeclasstopic.asp) array [This includes [XElement](http://msdn.microsoft.com/en-us/library/system.xml.linq.xelement.aspx) and [XNode](http://msdn.microsoft.com/en-us/library/system.xml.linq.xnode.aspx) array from .NET 3.5]
3. Enums
4. Types marked with DataContract or CollectionDataContract attribute
5. Types that implement [IXmlSerializable](http://msdn2.microsoft.com/en-us/library/system.xml.serialization.ixmlserializable.aspx)
6. Arrays and Collection classes including [List<T>](http://msdn2.microsoft.com/en-us/library/6sh2ey19(VS.80).aspx), [Dictionary<K,V>](http://msdn2.microsoft.com/en-us/library/xfhwa508.aspx) and [Hashtable](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionshashtableclasstopic.asp).
7. Types marked with [Serializable](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemserializableattributeclasstopic.asp) attribute including those that implement [ISerializable](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemruntimeserializationiserializableclasstopic.asp).
8. Types with none of the above attributes ([POCO](http://en.wikipedia.org/wiki/POCO)) but with a default constructor (can be non-public). [This is supported only from .NET 3.5 SP1]

Some types may implement more than one of the above programming models. In such cases a programming model is chosen based on its priority as given by the above list. For example, [Hashtable](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionshashtableclasstopic.asp) is a collection class but also implements ISerializable and it is serialized as a collection type. [DataSet](http://msdn2.microsoft.com/en-us/library/bwy42y0e(en-US,VS.80).aspx) implements both IXmlSerializable and ISerializable and it is serialized as IXmlSerializable.

<http://blogs.msdn.com/sowmy/archive/2006/02/22/wcf-serialization-programming-model.aspx>

Of these programming models, note that

* Implementing **ISerializable** is only possible in non-transparent code. (This is a huge limitation in the .NET Framework)
* Implementing **IXmlSerialzable** looks like a very useful back door, but it’s not clear it’s that easy.

## F#-Related Values to Consider w.r.t. Serialization

F# users may encounter interactions with serialization for the following types

* Tuples (whose closure would serialize correctly)
* F# unit values
* F# list values (whose closure would serialize correctly)
* F# ref values (whose contents would serialize correctly)
* F# option values (whose closure would serialize correctly)
* F# choice values (whose closure would serialize correctly)
* F# set values (whose closure would serialize correctly)
* F# map values (whose closure would serialize correctly)
* F# user-defined function values (whose closure would serialize correctly)
* F# user-defined structs (whose fields would serialize correctly)
  + Both generic and non-generic
* F# user-defined records (whose fields would serialize correctly)
  + Both generic and non-generic
  + Both with and without mutable fields
  + Both with and without extra getter/setter properties
* F# user-defined unions (whose fields would serialize correctly)
  + Both generic and non-generic
  + Of various shapes
    - Single case
    - Multiple case
    - With and without data
    - AllowNullLiteral attribute applies
  + Both with and without extra getter/setter properties
* F# user-defined classes (whose fields would serialize correctly)
  + Both generic and non-generic
  + With and without no-argument constructor
* F# user-defined delegates (whose closure would serialize correctly)
  + Both generic and non-generic
* F# user-defined enums

## Walkthrough for F# Serialization to JSON

**Recommendation: In the absence of other constraints, and if you need to use a framework serialization technoglogy, consider using JSON DataContract serialization for your F# serialization**

The examples in this section use JSON serialization. The advantages of XML and JSON are:

* “Plain old data” serializes in a reasonably straight-forward way
* The data is loosely coupled, e.g. a data structure that is serialized into a format can be deserialized as “matching” types.

JSON gives a simple human readable object format that intuitively matches the F# experience of data. Most F# values and types will work directly with XML and JSON serialization.

Here is an example:

#r "System.Runtime.Serialization.dll"

#r "System.ServiceModel.Web.dll"

open System.Runtime.Serialization.Json

open System.Runtime.Serialization

let serializeAndDeserialize (originalValue:'T) =

// Create a serializer, an output stream and serialzie the value

let serializer = DataContractJsonSerializer(typeof<'T>)

let out = new System.IO.MemoryStream()

printfn "input: %A" originalValue

serializer.WriteObject(out, originalValue)

// Get the serialized text

let text = out.ToArray() |> System.Text.Encoding.UTF8.GetString

out.Seek(0L, System.IO.SeekOrigin.Begin) |> ignore

printfn "serialized as: %s" text

let deserialziedValue = serializer.ReadObject out

printfn "deserialized as: %A" deserialziedValue

serializeAndDeserialize (1,2)

serialized as: {"m\_Item1":1,"m\_Item2":2}

deserialized as: (1, 2)

serializeAndDeserialize [| 1;2;3 |]

serialized as: [1,2,3]

deserialized as: [|1; 2; 3|]

serializeAndDeserialize 1.01

serialized as: 1.01

deserialized as: 1.01

let dict = new System.Collections.Generic.Dictionary<int,string>()

dict.Add(2, "two")

dict.Add(3, "three")

serializeAndDeserialize dict

serialized as: [{"Key":2,"Value":"two"},{"Key":3,"Value":"three"}]

deserialized as: seq [[2, two]; [3, three]]

**Recommendation: For your data types, use “Plain Old Serializable F# Data”, such as records, simple classes, ResizeArray (S.C.G.List), arrays and dictionaries.**

Plain Old Serializable F# Data means:

* Tuples
* Basic types (integers, floats, strings, DateTime, TimeSpan etc)
* Records
* Simple classes
* ResizeArray (S.C.G.List)
* Arrays
* Dictionaries (S.C.G.Dictionary)
* Set<\_>
* Map<\_,\_>
* Union types

However, be aware that

* You will need to add KnownType declarations to your union types in order to serialize them.
* You will not be able to serialize closures, object expressions or delegates. For these, you should use BinarySerialization.
* F# choice values may not be serialized with unless you register specific known types with the serializer.

For example:

type Record = { Id : int;

Values : int[] }

serializeAndDeserialize { Id = 1; Values = [| 1;2;3 |] }

serialized as: {"Id@":1,"Values@":[1,2,3]}

deserialized as: {Id = 1; Values = [|1; 2; 3|];}

type SimpleClass(id:int, names : int[]) =

member \_\_.A = id

member \_\_.B = names

serializeAndDeserialize (SimpleClass(3, [| 1;2;3 |]))

serialized as: {"id":3,"names":[1,2,3]}

deserialized as: SimpleClass(3, [| 1;2;3 |])

More complex types will also serialize correctly, e.g. try this one:

type Record2 =

{ Id: int;

Names: Set<int>;

OtherIds: Map<int,int>;

CallerIds: int[];

NumberLookupTable: Dictionary<int,int> }

**Recommendation: To use Union Types as Plain Old Serializable F# Data, add a KnownTypes attribute to your union types.** This code is boiler-plate, for example:

[<KnownType("KnownTypes")>]

type MyUnionType =

| Case1 of int \* int

| Case2 of string

static member KnownTypes() =

typeof<MyUnionType>.GetNestedTypes(BindingFlags.Public |||

BindingFlags.NonPublic)

|> Array.filter Reflection.FSharpType.IsUnion

For a generic union type, you must add an additional line:

[<KnownType("KnownTypes")>]

type MyUnionType =

| Case1 of int \* int

| Case2 of string

static member KnownTypes() =

typeof<MyUnionType>.GetNestedTypes(BindingFlags.Public |||

BindingFlags.NonPublic)

|> Array.filter Reflection.FSharpType.IsUnion

|> Array.map (fun ty -> ty.MakeGenericType [| typeof<'T> |])

**Recommendation: To precisely control the field names in your XML or JSON data, use a DataContract.**

For records, Plain Old Serializable F# Data will use names such as id@ for record names in the JSON format. Use a DataContract with DataMember targeted at the fields to remove the @ if needed. You can also use this attribute to write record types that match pre-existing JSON or XML formats.

[<DataContract>]

type R = { [<field: DataMember(Name="Ident")>]

Id : int;

[<field: DataMember(Name="Values")>]

Values : int[] }

serializeAndDeserialize { Id = 1; Values = [| 1;2;3 |] }

input: {Id = 1; Values = [|1; 2; 3|];}

serialized as: {"Ident":1,"Values":[1,2,3]}

deserialized as: {Id = 1; Values = [|1; 2; 3|];}

For classes, the serialization field names for classes used depend on the **internal** field names of a class, i.e. the names of the input and “let” variables that form the fields of the class. Avoid repetitions in these names, as a mangled name will then be used.

The serialization field names for classes can be controlled precisely by using a DataContract plus “let” bindings with a DataMember attribute targeting the fields. For example:

[<DataContract>]

type SimpleClass2(id:int, names : int[]) =

[<DataMember(Name="Ident")>]

let id = id

[<DataMember(Name="Names")>]

let names = names

member \_\_.A = id

member \_\_.B = names

serializeAndDeserialize (SimpleClass2(3, [| 1;2;3 |]))

serialized as: {"Ident":3,"Names":[1,2,3]}

deserialized as: SimpleClass2(3, [| 1;2;3 |])

Alternatively you may use a class with internal setters:

[<DataContract>]

type SimpleClass2(id:int, names : int[]) =

[<DataMember(Name="Ident")>]

let id = id

[<DataMember(Name="Names")>]

let names = names

member \_\_.A = id

member \_\_.B = names

For unions, in F# V1 you may not control the serialization names of union fields.

**Recommendation: When you need them, use the AutoSerializable(false) attribute to get POCO objects.**

POCO objects are “Plain Old C# Objects”. These are class definitions made up of

* private fields
* getter/setter properties
* a parameterless constructor.

They are commonly used where the implied serialization contract comes from the name of the type and the name of the fields. They are very useful because they are processed automatically by many tools (such as the serializers here), are very familiar to other .NET programmers, and can be used in partial trust.

To declare a POCO object in F#, add AutoSerializable(false). This turns off automatic serialization of field contents. Then ensure you have a parameterless constructor:

[<AutoSerializable(false)>]

type PocoObject(a:int, b : int[]) =

let mutable a = a

let mutable b = b

member \_\_.A with get() = a and set v = a <- v

member \_\_.B with get() = b and set v = b <- v

new () = PocoObject(0, [| |])

serializeAndDeserialize (PocoObject(3, [| 1;2;3 |]))

serialized as: {"A":3,"B":[1,2,3]}

deserialized as: FSI\_0035+PocoObject

For serialization purposes the setters need only be public if writing partial trust code.

This pattern only applies to F# class and struct types.

**Recommendation: Use NonSerialized for fields that should be omitted**

type Record3 = { Id : int;

Values : int[] ;

[<System.NonSerialized>]

mutable HashCodeCache : int option}

serializeAndDeserialize { Id = 1; Values = [| 1;2;3 |]; HashCodeCache=None }

serialized as: {"Id@":1,"Values@":[1,2,3]}

deserialized as: {Id = 1; Values = [|1; 2; 3|]; HashCodeCache = null;}

Be aware that the non-serialzied fields will be given the default value on deserialization. This is “another way” of getting default values in F#, and as such the fields should be marked mutable.

Note: the NonSerialized attribute can not yet be used on union type fields

**Recommendation: For Partial Trust code, limit your data structures.**

Somewhat counter-intuitively, partial trust code requires labelling more fields and properties as mutable. This allows you to deserialize the objects. For example:

* Mark all your record fields as mutable
* Do not use
  + F# sets or maps in your serialized data. Replace these with ResizeArray (S.C.G.List) and Dictionary. This will change the wire format of the data
  + F# union types in your serialized data. This is a restriction related to V1 of F#. Replace with equivalent records or objects aong with active patterns.
  + F# list values. Replace these with an array.
  + Tuple values in your serialized data. A .NET 4.0 restriction means that immutable types such as tuples may not be serialized or deserialized in partial trust.
* Make sure your classes have a parameterless default constructor and settable properties.

**Recommendation: Don’t make heavy weather of it. For very simple data sets, consider using bespoke serialization code if needed**. Learning and controlling serialization can take some investment**.** For example, consider manually printing your data as JSON.

## Sample JSON Wire Formats

Below are some examples, as they come out from the DataContractJsonSerializer

(1, 2) --> {"m\_Item1":1,"m\_Item2":2}

("2", 2) --> {"m\_Item1":"2","m\_Item2":2

None --> null

Some 1 --> {"value":1}

Some None --> {"value":null}

Some "1" --> {"value":"1"}

Some (1, 2) --> {"value":{"m\_Item1":1,"m\_Item2":2}}

{r1x = 1; r1y = 2;}   --> {"r1x@":1,"r1y@":2}

{r2x = 1; r2y = 2;}   --> {"r2x@":1,"r2y@":2}

type ImmutableRecord =

{ r1x: int; r1y:int }

type MutableRecord =

{ mutable r1x: int; mutable r1y:int }

ImmutableClass (3,4) --> {"a":3,"b":4}

MutableClass (3,4) --> {"a":3,"b":4}

type ImmutableClass(a:int, b : int) =

member \_\_.A = a

member \_\_.B = b

type MutableClass(a:int, b : int) =

let mutable a = a

let mutable b = b

member \_\_.A with get() = a and set v = a <- v

member \_\_.B with get() = b and set v = b <- v

set ["1"; "2"; "3"] --> {"serializedData":["1","2","3"]}

Map.ofList [ (2,"two"); (3,"three") -->

{"serializedData":[{"m\_Item1":2,"m\_Item2":"two"},

{"m\_Item1":3,"m\_Item2":"three"}]}

Giving a DataContract for a record type can clean up the names:

{r5x = 1; r5y = "2";}  --> {"x":1,"y":"2"}

[<DataContract>]

type ImmutableRecordWithPrivateDataContract =

{ [<field: DataMember(Name="x") >]

r1x : int;

[<field: DataMember(Name="y")>]

r1y : string;

}

[<DataContract>]

type MutableRecordWithPublicDataContract =

{ [<field: DataMember(Name="x") >]

mutable r5x : int;

[<field: DataMember(Name="y")>]

mutable r5y : string;

}

[<DataContract>]

type C(a:int, b : int) =

[<DataMember(Name="A")>]

let a = a

[<DataMember(Name="B")>]

let b = b

member \_\_.A = a

member \_\_.B = b

# Assembly Reference Resolution

The F# compiler and interactive defer to MSBuild reference resolution logic to find assemblies on disk. Many details of how MSBuild handles references resolution are covered elsewhere. This specification will focus on the particular aspects that are important or different in F#.

References are supplied to the compiler and interactive through the [**-reference**](#_--reference_<string>) flag. Additionally, interactive script files (which have the .fsx extension) support a **#reference** directive.

These references may be fully-qualified paths to assemblies on disk. In this case, no special rules are needed to find the assembly. Using full paths has the drawback that scripts aren’t easily moved from machine to machine. Also, when using the command-line compiler it may be more convenient to supply a short reference name and have the compiler locate the assembly.

Some examples of #reference use are:

// Fully qualified path to assembly name.

#reference @"c:\Windows\Microsoft.NET\Framework\v2.0.50727\System.dll"

// Simple assembly name. Found in .NET framework folder.

#reference @"System.dll"

// By long assembly name.

#reference @"System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"

// By short assembly name.

#reference @"System"

// Found via AssemblyFoldersEx registry key lookup.

#reference @"CrystalDecisions.Shared"

The reference resolution logic looks through a specific predefined set of locations. The set of locations considered different in compiler and interactive.

## Compiler Search Locations

1. Treat the name as a fully qualified file name
2. Look in the folder containing the current project file.
3. Look in the directory with fsc.exe in it.
4. Look in the target .NET framework directory.
5. Look in directories registered under AssemblyFoldersEx registry key.
6. Look in directories registered under AssemblyFolders registry key.
7. Look in the project output directory.

## Interactive Search Locations

1. Treat the name as a fully qualified file name
2. Look in directories specified by [#I](#_#I)
3. Look in the folder containing the current script (.fsx) file.
4. Look in the directory with fsc.exe in it.
5. Look in the target .NET framework directory.
6. Look in directories registered under AssemblyFoldersEx registry key.
7. Look in directories registered under AssemblyFolders registry key.
8. Look in the Global Assembly Cache.

## F# Attributes

<Put something here>

### OCamlCompatibility

<Put something here>

# Compiler Architectural Breakdown

Key phases

|  |  |  |  |
| --- | --- | --- | --- |
| Phase | Kind | Files | Comments |
| Input preparation etc. |  |  |  |
| * Input preparation, command-line driver |  | fsc.fs, fscmain.fs | Allowing GC of large data structures is important |
| * Interactive driver |  | fsi\fsi.fs | Messy. |
| * Compiler options |  | build.fs  fscopts.fs | A bit messy |
| Lexing |  |  |  |
| * Lexing | Lex-tyle tokenizer | lex.fsl |  |
| * Lex helpers |  | lexhelp.fs |  |
| * Lex table interpreter |  | fsppack\prim-lexing.fs |  |
| * Lex buffers |  | fsppack\prim-lexing.fs |  |
| * Lexer table generator |  | fslex\\* |  |
| Lex Filtering | Token stream processor (state machine) | lexfilter.fs |  |
| Parsing |  |  |  |
| * Parser | LALR(1) | pars.fsy |  |
| * Parser table interpreter |  | fslib\prim-parsing.fs |  |
| * Parser table generator |  | fsyacc\\* |  |
| * Untyped Abstract Syntax Trees | Data structure | ast.fs |  |
| * Position markers | Data structure | range.fs |  |
| Typed Abstract Syntax Trees |  |  |  |
| * Data Structure | Data structure | tast.fs | Possible elimination of some constructs? |
| * Global “compiler-known” constants | Data table | env.fs |  |
| * Basic common TAST operations w.r.t. global constants |  | tastops.fs | Debug printing should be #DEBUG |
| * Pickle to/from F# binary metadata resource |  | pickle.fs |  |
| * Infos representing uses of types and members |  | infos.fs | Possible value in using these directly in TAST |
| Import .NET to TAST |  |  |  |
| * Abstract IL representation of .NET metadata | Data structure | absil\il.fs  absil\il.fsi |  |
| * Read .NET metadata |  | absil\ilread.fs |  |
| * Convert .NET metadata to TAST types and type definitions |  | import.fs |  |
| * On-demand loader for .NET binaries |  | build.fs |  |
| Type checking, inference and elaboration of AST to TAST |  |  |  |
| * Drive typechecking of a sequence of files |  | build.fs | Should be in tc.fs?  Should be less linear, more incremental? |
| * Constraint solving |  | csolve.fs | Uses imperative mutation and backtracking |
| * Name resolution |  | nameres.fs | Uses tables currently in TAST. These should be factored out of the TAST? |
| * Checking attributes |  | tc.fs |  |
| * Checking types |  | tc.fs |  |
| * Checking expressions |  | tc.fs |  |
| * Checking module and type declarations |  | tc.fs |  |
| * Pattern match compilation |  | patcompile.fs | Complex algorithm |
| * Method overload resolution |  | csolve.fs | Uses backtracking solving |
| * Resolution of computation expressions |  | (currently tc.fs) |  |
| * Representation of implicit construction |  | (currently tc.fs) | Representation choices could be made elsewhere? |
| * Auto-generation of hash/eq/compare |  | augment.fs |  |
| * Format string resolution |  | format.fs |  |
| * Closing the inference scope |  | (currently tc.fs) |  |
| * Signature checking |  | tc.fs, typrelns.fs |  |
| * Post-inference Checking |  | unsolved.fs, check.fs |  |
| Optimization |  |  |  |
| * Optimization summaries | Data structure | opt.fs | These pushed across file/assembly boundaries and go in a binary resource |
| * Optimization via Simplificiation and Inlining |  | opt.fs |  |
| * Optimization via Lambda Lifting (“Top Level Representations”) |  | tlr.fs  lowertop.fs |  |
| * Optimization via Detupling |  | detuple.fs | Delete? |
| Quotation binary data generation |  |  |  |
| * quotation trees | Data structure | sreflect.fs  fslib\quotations.fs |  |
| * Generation of quotation trees from TAST |  | creflect.fs |  |
| * Generation of binary representation from quotation trees |  | sreflect.fs  fslib\quotations.fs |  |
| Abstract IL tree generation |  |  |  |
| * Generation of Abstract IL types, code and definitions from TAST |  | ilxgen.fs |  |
| * Module generation buffers | Data structure | ilxgen.fs |  |
| * Code generation buffers | Data structure | ilxgen.fs |  |
| * Generation of Abstract IL code trees from linear opcode sequences |  | absil\il.fs |  |
| * Helpers for Abstract IL trees |  | absil\il.fs |  |
| * Debug printing for Abstract IL trees |  | absil\ilprint.fs | #DEBUG only? |
| * Elimination of ILX Closures |  | ilx\pubclo.fs | Change to a TAST transformation? |
| * Elimination of ILX Discriminated Unions |  | ilx\cu\_erase.fs | Change to a TAST transformation? |
| Code emit |  |  |  |
| * Binary writing |  | absil\ilwrite.fs | Performance an issue here. Make incremental? |
| * Reflection emit |  | absil\ilreflect.fs |  |

## Miscellaneous Architectural Questions

Set of questions arising from design walk through with Dmitry and Anar (Nov 2008)

* F# does both qualified and unqualified name resolution during type checking. Is it feasible to do unqualified name resolution (as far as method groups, i.e. no overload resolution) earlier, as a pre-phase? Would this buy as anything?
* Pattern match compilation is done during type checking. This means pattern match compilation is run during background type checking in visual studio? Could we do it later?
  + No, not really, since we have to run pattern match compilation to generate the errors and warnings for incomplete and redundant matches. This can't really be done separately from compiling the pattern to a TAST

# Compiler Performance Analysis

This section includes techniques and analysis of recurring patterns in compiler performance

Note that fsc.exe compiler throughput performance does NOT necessarily translate directly to Visual Studio Language Service performance – the latter is *reactive* performance and response time is critical rather than throughput.

## Methodology:

* Using staging Retail build
* VS 2008 -> Analyze -> Profile -> New Performance Session
* Add a new target “fsharp\Retail\bin\fsc.exe”
  + src\tests\fsharp\bootstrap.bat gives command line arguments
  + C:\fsharp\staging\src\fsharp\FSharp.Compiler is the working directory
  + add --pause as an extra command line argument
* Start with profiling off
* Step through to the phase of the compiler you want to profile
* Start profiling
* After phase ends, hit ctrl-C
* Analyze results, typically starting with fringe “exclusive functions” in the summary and working up the caller/callee display to most meaningful “inclusive function”

## Analyzing Results

Performance can be analyzed from several levels

* **Phases.** The compiler phases (parse, typecheck etc.)
* **Algorithms.** The compiler algorithms (generalization, TLR, etc.)
* **Representations.** The compiler data representations and core operations (TAST, AST, free variable collecting etc.)
* **Library Implementation.** Low level F#/BCL library operations (creating set/Map/Dictionary/List/Tuple, looking up Set/Map/Dictionary etc.)
* **Code Gen.** Low level F#/CLR operations (tailcalls, is-instance, castclass, GC, allocation, fetching static fields etc.)
* **Physical.** X86 code, caches, branch predication, page faults etc.

## Primary Phases

Results of analysis, Sept 2009, (based on bootstrap compilation of FSharp.Compiler.dll):

Parse Inputs 7%

Typecheck 30%

Optimizations 25%

TAST --> ILX 10%

ILX --> IL 4%

Write Binary 7%

Write PDB 17%

## Parsing/Lexing phase

Most the time here is spent in the lex filter

## Typecheck phase

### Typecheck phase - algorithmic aspects

**Opening modules**: Opening a module adds the symbols for the module to various environment tables

**Computing ungeneralizable type variables:**

Each “let” bindings triggers a call to ComputeUngeneralizeableTypeVariables

**Checking Method Applications and Resolving Method Overloading**:

See TcMethodApplication

**Resolving names for the dot-notation for generic types and other lookup cases which we do not cache**

We use various caches to amortize the cost of looking up names such as “obj.Property”. At the moment these caches are only used when the type being analyzed is precisely a TType\_app(tcref,[]) node. This means we are recomputing the lookups often for other cases.

We should be able to amortize these costs for other lookups, notably on closed types when no backtracking is going on (the normal case)

### Typecheck phase - representation aspects

**Collecting free variables**

Free variables are collected for various purposes, e.g. generalization and checking “escape” conditions

**Expanding type abbreviations**

We maintain type abbreviations in the TAST trees and repeatedly re-expand them as we traverse the type nodes. This appears as a significant cost, though I don’t know how to amortize this as yet

| Function Name | Inclusive Samples | Exclusive Samples | Inclusive Samples % | Exclusive Samples % |
| --- | --- | --- | --- | --- |
| |  | | --- | | Tastops.apply\_tycon\_abbrev(class Tast/typ,class Tast/Entity,class FSharpList`1<class Tast/typ>) | | 266 | 8 | 3.43 | 0.10 |

### Typecheck phase – low level and CLR costs

**Building sets and maps**

Sets are used when collecting free variables, Maps are used for the “environments”

**Allocating and Garbage Collecting**

e.g. typical manifestation:

**Current function**

| Function Name | Inclusive Samples | Exclusive Samples | Inclusive Samples % | Exclusive Samples % |
| --- | --- | --- | --- | --- |
| |  | | --- | | Alloc(unsigned int,int,int) | | 984 | 1 | 12.70 | 0.01 |

Of which this much is contributed by triggered GCs of generation #1:

**Functions that were called by WKS::GCHeap::SuspendEE(enum GCHeap::SUSPEND\_REASON)**

| Function Name | Inclusive Samples | Exclusive Samples | Inclusive Samples % | Exclusive Samples % |
| --- | --- | --- | --- | --- |
| |  | | --- | | WKS::gc\_heap::gc1(void) | | 747 | 0 | 9.64 | 0.00 |

And these callsites are indications of allocation hot points:

**Functions that called FastAllocateObject(class MethodTable \*)**

**Write Barriers**

The CLR calls a write barrier helper when allocating common objects such as tuples and Set nodes, e.g. typical manifestation:

**Comparing strings (when looking up Namemap’s),**

e.g. typical manifestation:

**Fetching the “empty list” object**

The empty list is stored in a static field in the List<T> class. Simply fetching this object from appropriate tables takes considerable time:

**Current function**

Most of this stems from calls to “map”. Surprisingly looking up the static field often ends up making non-trivial calls – this reflects the CLR implementation of static field access, but it is surprising to see some of the C++ functions on the list below:

**Functions that were called by FSharpList`1.get\_Empty()**

## TAST 🡪 ILX phase

**Building the structured ILX “code” object from the linear sequence of instructions.**

This takes nontrivial time. It is likely the entire ILX -> IL phase can be removed in favour of localized transformations on the instructions as we emit them in ILXGEN. Alternatively ILXGEN could emit a structured “code” object directly

**Current function**

**Keeping track of the IL stack**

There is a lot of garbage generated in this phase, largely to keep track of the types on the IL stack in case we need to flush them. This is a bit unfortunate since it is needed very rarely.

Using a Map to store the storage for values may be sub-optimal – a hash table may be better, e.g. creating the maps is 4%

**Current function**

And looking them up is 2.5%

**Current function**

**Generating Abstract IL types from TAST types**

It feels like it should be possible to amortize this cost most of the time

**Current function**

## ILX 🡪 IL phase

## Write Binary phase

### Write PDB

Writing large PDBs takes a lot of time, this may be due to some inherent inefficiency in the PDB Writer API, or we may be using it in a way that is hitting some worst case behaviour.

## Memory Usage of Compiler and Language Service

The F# compiler and language service use a lot of memory. Reducing this should in theory give major performance gains, though getting really wins seems surprisingly difficult.

### Taking a memory dump using windbg and SOS profiling

/// Connect WINDBG to the process

windbg.exe -p <process-id>

/// Load the SOS debugging DLL

.load C:\Windows\Microsoft.NET\Framework\v4.0.30319\SOS.dll

/// Show a histogram of the live objects by type/size

!dumpheap -stat

/// Show a histogram of the string objects by type/size/contents

!dumpheap -strings

/// Show the big byte arrays (> 5MB)

!dumpheap -type System.Byte[] -min 5000000

/// Show the addresses of first 100 ‘Op’ objects in the heap

!dumpheap -type Tast+Expr+Op

/// Show a trace of the GC roots for a particular object

!gcroot -nostacks 33719324

Here’s a typical analysis of this data:

I did a quick VS memory dump using SOS while editing the F# compiler itself. Here are the top entries while editing the compiler itself. Quick notes:

* I think we’re using more memory, because we’re now capturing AST expression trees

#instances total size

193082 3089312 Ast+SynExpr+Ident

194042 3104672 FSharpList`1[[Ast+Ident]]

232094 3713504 Ast+SynConst+UInt16

157903 5052896 Ast+SynExpr+App

243969 8782884 Ast+SynExpr+Seq

* I’m surprised to see so many TAST expression tree nodes, I don’t expect that. We shouldn’t be capturing those. I don’t know where these are coming from I’m going to look into it a bit

95548 3439728 Tast+Expr+App

153354 4907328 Tast+Expr+Op

364572 5833152 FSharpList`1[[Tast+Expr]]

* Here is a needless allocation – we could put this in TcEnv

#instances total size

146814 2936280 Infos+AccessorDomain+AccessibleFrom

* It occurs to me that TyparData objects could be reduced in the size in the common case where they are inference variables. I’ve added this note:

// MEMORY PERF: TyparData objects are common. They could be reduced to a record of 4-5 words in

// the common case of inference type variables, e.g.

//

// TyparDataCommon =

// typar\_details: TyparDataUncommon // null indicates standard values for uncommon data

// typar\_stamp: Stamp

// typar\_solution: TType option

// typar\_constraints: TyparConstraint list

// where the "common" settings are

// kind=TyparKind.Type, rigid=TyparRigidity.Flexible, id=compgen\_id, staticReq=NoStaticReq, isCompGen=true, isFromError=false,

// dynamicReq=TyparDynamicReq.No, attribs=[], eqDep=false, compDep=false

### Top Entries By Type When Typechecking

Taksn 20/4/2011, interrupting the bootstrap of the compiler at an arbitraty point

787100 Tast+EntityData

791580 Tast+DecisionTreeTarget

803264 Func`1[[FSharpList`1[[AbstractIL.IL+ILAttribute]]]]

831568 UInt16[]

907040 Tast+EntityRef

921404 Dictionary`2+Entry[[PrettyNaming+NameArityPair],[Tast+EntityRef]][]

944768 Dictionary`2+Entry[[String],[FSharpList`1[[Tast+EntityRef]]]][]

1054752 FSharpList`1[[Ast+SynPat]]

1107920 Ast+SynPat+LongIdent

1122208 Tast+FreeVars

1143968 FSharpList`1[[Tast+Val]]

1195776 Ast+SynExpr+LongIdent

1204940 Tast+Binding

1289680 Tast+Expr+Const

1296672 Tast+Val

1320980 Tast+TType+TType\_fun

1351168 Tast+Expr+Let

1399740 Tast+TType+TType\_app

1402900 Ast+SynPat+Wild

1455216 Microsoft.FSharp.Core.FSharpOption`1[[Tast+TType]]

1544976 Tast+Expr+Lambda

1563156 Ast+SynExpr+Paren

1590504 Int32[]

1617448 Ast+SynBinding

1621228 Object[]

1663308 Object

1760340 Dictionary`2+Entry[[String],[Nameres+Item]][]

2043684 Ast+SynPat+Named

2311408 Tast+ValRef

2507712 Ast+SynExpr+Ident

2557680 Tast+TType+TType\_var

2557680 Tast+Typar

2610288 Tast+Expr+App

2616896 FSharpList`1[[Ast+Ident]]

3349536 SetTree`1[[Tast+Val]]

3434080 FSharpList`1[[Tast+TType]]

3526496 Tast+Expr+Op

3713504 Ast+SynConst+UInt16

4104768 Ast+SynExpr+App

4286016 FSharpList`1[[Tast+Expr]]

5305268 String

5305468 Tast+Expr+Val

6245208 Ast+SynExpr+Const

7033620 Tast+TyparData

8249620 Ast+Ident

8718264 Ast+SynExpr+Seq

9508928 Tast+ValData

### Example dump during ILX Generation

077f1c54 64886 1038176 Types+IlxUnionSpec

077c10a4 65066 1041056 Tast+DecisionTree+TDSuccess

077c1764 24548 1080112 Tast+Expr+Match

077f4eec 68865 1101840 IL+ILInstr+I\_other

68702938 18885 1119292 Int32[]

077c1454 57716 1154320 Tast+DecisionTreeTarget

0772de88 15012 1261008 IL+ILTypeDef

00592034 106561 1278732 LazyExtensions+Create@5647[[FSharpList`1[[IL+ILAttribute]]]]

077f4544 53481 1283544 IL+ILInstr+I\_call

077f3e2c 111492 1337904 IL+ILAttributes

077f428c 84316 1349056 IL+ILInstr+I\_ldarg

052ba034 86115 1377840 FSharpList`1[[Int32]]

0759b7d4 87857 1405712 FSharpList`1[[Tast+Val]]

077f4334 89459 1431344 IL+ILInstr+I\_ldloc

077c148c 71632 1432640 Tast+Binding

077f734c 45723 1463136 IL+ILParameter

0772806c 95684 1530944 Tast+UnionCaseRef

0772db14 24910 1594240 IL+ILMethodDef

077c2d60 51563 1650016 Tast+FreeVars

07713620 138736 1664832 Tast+Val

077f601c 104074 1665184 IL+ILCode+ILBasicBlock

0772dd84 104074 1665184 IL+ILBasicBlock

077c16f4 52061 1665952 Tast+Expr+Let

077c0248 119691 1915056 Tast+TType+TType\_var

07727aa8 119691 1915056 Tast+Typar

077c156c 69625 1949500 Tast+Expr+Const

07727b60 82117 1970808 IL+ILTypeRef

0768499c 124221 1987536 FSharpList`1[[AbstractIL.IL+ILCode]]

0773540c 73407 2055396 IL+ILSourceMarker

077c1614 44997 2159856 Tast+Expr+Lambda

07674ac8 111492 2229840 System.Lazy`1[[FSharpList`1[[IL+ILAttribute]]]]

077c1684 63636 2290896 Tast+Expr+App

07714d64 151780 2428480 Tast+ValRef

07735224 122702 2454040 IL+ILMethodSpec

077334a8 81210 2598720 IL+ILMethodRef

68704944 34151 2733576 Byte[]

076aceb0 106562 3409984 Func`1[[FSharpList`1[[IL+ILAttribute]]]]

686ff5e8 301829 3621948 System.Object

077c01d8 198418 3968360 Tast+TType+TType\_fun

077f3060 248465 3975440 IL+ILType+Boxed

0773371c 272167 4354672 IL+ILTypeSpec

077c0168 229387 4587740 Tast+TType+TType\_app

077bf89c 119691 5266404 Tast+TyparData

077105f4 221809 5323416 SetTree`1[[Tast+Val]]

07731238 177446 5678272 Tast+Expr+Op

07464ab4 375913 6014608 FSharpList`1[[Tast+Expr]]

0759aea8 411770 6588320 FSharpList`1[[Tast+TType]]

07680d7c 412860 6605760 FSharpList`1[[IL+ILType]]

686b6c28 201110 6927612 System.Object[]

077c15a4 281988 7895664 Tast+Expr+Val

686ff9ac 172837 10144856 System.String

077285e8 138736 12208768 Tast+ValData

# Coding Conventions in the F# Compiler

## Abbreviations used in the F# Compiler Code

|  |  |  |
| --- | --- | --- |
| Abbreviation | Terminology | Notes |
| acc | Accumulate  Accumulating parameter |  |
| anon | Anonymous | e.g. anonymous type variable “#Foo” |
| aenv | Alpha-equivalence Environment | Map variable names to type variable names for checking equivalence w.r.t. renaming of type variables |
| app | Application | e.g. type application “Foo<int>” or expression application “f x” |
| apinfo | Active pattern info |  |
| apelem | Active pattern element |  |
| asm | Assembly code | i.e. a sequence of Abstract IL instructions |
| attr, attrib, cattrs | Attributes | usually referring to F# representations of custom attributes, occasionally referring to Abstract IL representations of custom attributes |
| cenv | Compiler Environment | Tables, often mutable, global to a phase of the compiler |
| ccu | Compilation unit | = F# view of a DLL, EXE or the current assembly being compiled |
| ccuthunk | Cross-compilation unit thunk | Thunk that can be fixed up to refer to another compilation unit |
| constr | Constructor | usually referring to a discriminated union case label |
| conj | Conjunction | “and”, e.g. of patterns |
| cpath | Compilation path | The fully qualified path to an entity, see also pubpath which is very similar |
| deref | Dereference | e.g. dereference a value, type or module reference |
| disj | Disjunction | “or”, e.g. of patterns |
| env | Environment | Immutable tables propagated top-down through a walk of a tree |
| exn | Exception |  |
| exnc | Exception constructor | A type constructor representing an F# exception definition, e.g. exception Fail = E of int |
| err | Exception | representing an error |
| dtree | Decision Tree | A data structure used to represent compiled pattern matching |
| fsobjmodel | F# Object Model | F# class, struct, delegate, interface definition |
| fun | Function | e.g. “Type\_fun” for function type |
| funion | Finite union | A discriminated union |
| g | Globals | Table of type-checker globals |
| generalize | Generalize | Convert an entity (normally a type constructor) to its generic type, e.g. List<\_> to List<'a> where 'a is the type parameter of the generic type definition. |
| id, ident | Identifier | i.e. name/range pair |
| il | Intermediate Language | Something relating to Abstract IL |
| inst | Instantiation (n.)  Instantiate (v.)  Instance | Instantiation (see tinst/tyargs)  Apply a variable to type/value mapping to an entity  Entities related to instance checks, e.g. “isinst” |
| isext | Is Extension | Is something an extension member? |
| lid | Long Identifier | i.e. identifier list for A.B.C |
| lvalue | Left-value | much in the sense as C, i.e. entity appearing on the left of an assignment E <- V. |
| m | Range mark | , i.e. a position pair in original source code |
| mk | Make |  |
| mksyn | Make syntactic | e.g. make an AST node |
| mmap | Multi-map | Immutable map one-to-many |
| modul | Module | Data structure representing an F# module or namespace fragment  Occasionally a .NET module (DLL) in a potentially-multi-module .NET assembly |
| mref,modref | Module reference | Reference to an F# module or namespace fragment  Occasionally an Abstract IL reference to a .NET module (DLL) in a potentially-multi-module .NET assembly |
| mtyp | Module type | Data structure representing the contents of an F# module relevant to type checking |
| mut | Mutable |  |
| nenv | Name Resolution Environment | Tables that guide the process of resolving names (e.g. “map” or “System”) to semantic objects |
| nlr | Non-local reference | Reference to an entity potentially in another compilation unit |
| opt | Optional | e.g. optional argument  e.g. value of type “Option” |
| opt | Optimize |  |
| pat | Pattern |  |
| plid | Partial long identifier | e.g. Hello.Wor where “Wor” is incomplete |
| ptc | Partial type check | Resolve a Partial long identifier to a set of possible completions |
| pubpath | Public path | What is the fully qualified name to an entity, if any |
| qname | Qualified Name | A fully qualified .NET assembly name for a compilation unit, e.g. FSharp.Core, Version=1.9.3.4, ... |
| recd | Record |  |
| repr | Representation | e.g. the r.h.s. of a type definition “type x = A | B” |
| rfield | Record field |  |
| rfref | Record field reference | A pair containing a reference to a type constructor and a record field name |
| scoref | Scope Reference | Abstract IL value representing the .NET metadata we need to emit to reference another assembly |
| spat | Simple pattern | a kind of AST node for some places where only limited patterns are allowed or where complex patterns have been removed early via desugaring |
| strip | Strip | e.g. strip lambdas off a lambda expression, or repeatedly reduce type abbreviations |
| subst | Substitute | Replace variables by types/values |
| tau | “Tau type” (the greek letter tau) | Traditionally the types of generic values etc. have been written ∀α1...αn. τ where α1...αn are the generic type parameters. In F# syntax this might look more like <α1...αn> τ. Either way the type that’s left after you strip off the generic type parameters is called “tau” |
| tc | Type check |  |
| tcaug | Type constructor augmentation | Extra information about a named type definition nearly all relating to the “object model”, e.g. members |
| tlr | Top level representation | Lambda lifting optimization |
| tp(s), typar(s), tyvar(s), tyv(s) | Type parameter(s) |  |
| typar\_inst, tpenv | Type parameter instantition/environment | A map from type variables to types |
| tpc(s) | Type constraints (s) |  |
| tinst, tyargs | Type instantiation | <int>, <int,string> |
| tyapp | Type application | Foo<int> |
| ty,typ | Type | Abstract IL and F# types |
| tycon | Type constructor | i.e. named type definition such as List<\_> |
| tcref | Type constructor reference | A reference to a named type definition such as List<\_> |
| tyenv | Type environment | A pair of maps mapping   * type variables to types (a type parameter instantiation) * type constructors to type constructors |
| tyfunc | Type function | A non-function value defined with explicit type parameters let empty<'a> = ... |
| uconstr | Union constructor | A discriminated union case label |
| ucref | Union constructor reference | A pair containing a reference to a type constructor and a discriminated union case name |
| val, vspec | Value  Value Specification | An F# value x or an F# member |
| vrec | Value recursive | A reference to a local value, a value in another module or a value in another assembly |
| vref | Value reference | A reference to a local value, a value in another module or a value in another assembly |

# F# Code: Optimized v. Debug

## Interaction with Debug Tools

### Just My Code, Tailcalls and Stepping (VS2008)

*To complicate the matter further, the presence of tailcalls in the F# library messes up JustMyCode debugging of F# code as well (it also messes up AllCode). If there is even one tailcall on a execution path through a library call then step-through debugging fails to do anything sensible. For example*

*let TestFunction15(inp) =*

*let x = inp+1*

*[1;2;3] |> List.map (fun x -> x + 1)*

*This steps perfectly for a “user defined, no-tailcall” List.map and fails to step in any kind of sensible fashion for the F#-library-defined List.map (which uses an internal tailcall to call the real List.map implementation in Locals.List.map)*

*The F# library we ship is, in effective, giving a bad debug experience in conjunction with VS 2008.*

## Debug v. Optimized Code Generation

Our product does at least 4 things differently in debug/optimized code generation in ILXGEN. These are documented below.

(1) Shadows locals are emitted in “main” method for toplevel immutable locals

// decide whether to use a shadow local or not

let useShadowLocal =

cenv.generateDebugSymbols &&

not cenv.localOptimizationsAreOn &&

not v.IsCompilerGenerated &&

not v.IsMutable &&

// Don't use shadow locals for things like functions which are not

// compiled as static values/properties

IsCompiledAsStaticProperty cenv.g v

(2) FeeFee breakpoints for hidden code

// For debug code, emit FeeFee breakpoints for hidden code, see http://blogs.msdn.com/jmstall/archive/2005/06/19/FeeFee\_SequencePoints.aspx

member cgbuf.EmitStartOfHiddenCode() =

if mgbuf.cenv.generateDebugSymbols && not mgbuf.cenv.localOptimizationsAreOn then

let doc = mgbuf.cenv.g.memoize\_file (file\_idx\_of\_range m)

codebuf.Add(FeeFeeInstr mgbuf.cenv doc);

(3) Different code for pattern matching

// In debug mode push all decision tree targets to after the switching

let isTargetPostponed =

if cenv.localOptimizationsAreOn then

GenDecisionTreeTarget cenv cgbuf stackAtTargets targetIdx targetInfo sequel;

false

else

CG.EmitInstr cgbuf [] (I\_br targetMarkBeforeBinds.CodeLabel);

true

(4) Reusing locals

// Get an index for the local

let j =

if cenv.localOptimizationsAreOn

then cgbuf.ReallocLocal((fun i (\_,ty') -> not (IntMap.mem i eenv.liveLocals) && (ty = ty')),ranges,ty)

else cgbuf.AllocLocal(ranges,ty)

# FsYacc Notes

This topic is about the FsYacc tool in the F# Power Pack.

## Getting better error messages out of fsyacc generated parsers

Here’s the approximate magic we use in the compiler:

1. Try defining **parse\_error** in the code section of your .fsy file and check you get a different message. This is for very simple cases, e.g. to report a line number
2. You can then instead define **parse\_error\_rich**, and analyze the “context” given to you. The important fields are ctxt.ShiftTokens, ctxt.CurrentTokens, ctxt.ReducibleProductions. These are integers which need to be mapped across, e.g. approx this:

sprintf "Unexpected token %A in one of %A. Expected one of %A"

ctxt.CurrentToken

(List.map Parser.prodIdxToNonTerminal ctxt.ReducibleProductions)

(List.map Parser.tokenTagToTokenId ctxt.ShiftTokens)

# History of the F# Compiler Codebase and Project

1999.

* “Model implementation of the .NET verifier”, written in OCaml by Don Syme, Andy Gordon and interns.
* Metadata readers read ILDASM format and attempted to use the COM reader APIs
* OCaml-COM interop a major pain point

2000-2001.

* Abstract IL library developed from the compiler codebase. Released as binary code.
* ILX project. Aim to “add extra functional programming constructs to the .NET IL”, in the contxt of “Project 7”. Closures, generics and discriminated unions added as extra constructs in Abstract IL.
* Elimination of ILX Generics by rewriting (now deleted, was pp\_erase.fs and nupp\_erase.fs)
* Elimination of ILX Closures by rewriting “pubclo.fs”
* Elimination of ILX Discriminated Unions by rewriting “cu\_erase.fs”
* .NET Generics the major spinoff of this project (dsyme, akenn).

2002-2003

* F# project starts. Aim to “build a .NET compiler for a practical language in under 10,000 lines”.
* Parser “pars.fsy”, typechecker “tc.fs”, code generators “ilxgen.fs” (dsyme)
* Add hash/compare/eq automaticallt “augment.fs” (dsyme)
* Binary reader/writer of .NET metadata “ilread.fs, ilwrite.fs” (dsyme)
  + implemented using Abstract IL library
* Project landmarks:
  + First release of “0.5” of F# around Dec 2002.

2004

* “printf” implemented via code generation (dsyme)
* Many libraries implemented (dsyme)
* Binary metadata pickler “pickle.fs” (dsyme)
* Method overload resolution (dsyme)
* Simplifier optimizer “opt.fs” (dsyme)
* Project landmarks:
  + First cross-compilation of Abstract IL
  + First cross-compilation of significant apps (e.g. Zapato theorem prover)
  + Dominic Cooney uses F# version of Abstract IL for internship projects

2005

* F# Visual Studio mode, “service.fs” and all the C++ code (dsyme)
* TLR and Detuple optimizations (jamarg)
* First version of the F# object model and subtype-aware inference (dsyme).
  + “Hacked” into the codebase in ways that were later highly problematic
* Microsoft.FSharp.Reflection, “reflect.fs” (dsyme)
* F# Interactive, “ilreflect.fs” (jamarg)
* fsyacc (dsyme)
* fslex (dsyme)
* Project landmarks:
  + First signs of serious interest and uptake

2006

* F# Quotations “quotations.fs”, “creflect.fs”, “sreflect.fs” (dsyme)
* FLinq (dsyme)
* Explicit passing of globals throughout compiler “env.fs” (dsyme, jamarg)
* Active patterns in “patcompile.fs” (gneverov, dsyme)
* Implicit construction of classes in “tc.fs” (jamarg)
* Project landmarks:
  + Frist DevDiv presentation in Feb

2007

* Parser cleanup and disambiguations (jamarg)
* Cleanup of TAST “tastops.fs” (dsyme, jamarg)
* Cleanup of typechecker “nameres.fs”, “csolve.fs”, “typrelns.fs”, “infos.fs” (dsyme, jamarg)
* Deletion of significant amounts of Abstract IL code (dsyme)
* Creating of “staging” branch
* Project landmarks:
  + Productization decision

2008

* Byref checking “check.fs” (jamarg)
* Language service rewrite “servicem.fs” etc. (jomof)

# F# Async Ideas

List of features we might consider adding in later versions of F#:

1. Async state machines
2. AsyncResultCell – the implementation of a Future for asyncs
3. Task interop
4. IAsyncDisposable
5. ThreadAffinityContext
6. asyncSeq
7. Enabling async hops across domains
8. Making MailboxProcessor MarshalByRef

## Async State Machines

See allocation-free-async-workflows.docx

## AsyncResultCell

We ship AsyncResultCell in PowerPack today, and we have an internal implementation of the same in FSharp.Core. The F# async version of the Future proved to be extremely useful in many scenarios – we must make it available to F# programmers out of the box.

## Task Interop

TBD

## IAsyncDisposable

So in our today’s implementation of asynchronicity in a language service, we use this pattern:

    member info.GetDeclarations((line,colAtEndOfNames),lineStr,names:NamesWithResidue,tokenTag:int) =

        async {

            do! switchToReactorThread

            return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

        }

And at the call site:

async {

    let! decls = req.ResultScope.GetDeclarations(req.View, req.Line, req.Col, req.TokenInfo, reason)

    do! Async.SwitchToContext UIThread.TheSynchronizationContext

    …

}

This is an example of code that is correct (well, there is a bit of exception handling issue, but let’s ignore that for the sake of this discussion), and *in the given circumstances is actually an “optimal” one*, but is an architecture nightmare. There are two reasons to that, one has to do with suboptimal story for thread affinity in F# asyncs, and one with the design disaster that is SynchronizationContext.Current. Let’s go over these in order and examine the solutions.

What the code above wants to do (and it is in fact quite typical scenario) is:

* Execute “DeclarationSet(scope.GetDeclarations(),…)” asynchronously on *reactor* thread.
* Get back on the UI thread

The way the code is written above, the continuation of the async that GetDeclarations returns *will continue to run on  reactor thread*. That is why we need Async.SwitchToContext at the call site. Obviously, GetDeclarations shouldn’t let the continuation run on reactor thread – it wants the continuation to return to the “thread”, or “context” where it came from. The abstraction we use for that today is SynchronizationContext, so the code you will want to write looks like this:

        async {

            let sc = System.Threading.SynchronizationContext.Current

            try

                do! switchToReactorThread

                return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

            finally

                do! Async.SwitchToContext sc

        }

Which has two issues:

1. It is really ugly and does not convey the intent well
2. You cannot actually write that today, because “finally” blocks do not let you bind inside of them!

Thanks to Tomas, we have a suggestion of how to deal with this: “use!” and IAsyncDisposable to the rescue:

        async {

            use! \_ = Async.ReactorThread

          return DeclarationSet(scope.GetDeclarations line lineStr colAtEndOfNames names)

        }

where Async.ReactorThread is an Async<IAsyncDisposable> which captures the sync context, schedules the continuation onto reactor thread and in Dispose async schedules continuation back onto the original sync context.

Our general guidance says that all async APIs should return back to SynchronizationContext they have been called on. This style of programming gives F# programmer a really nice tool for following that guidance.

The callsite will also look rather nice:

async {

       use! \_ = Async.SynchronizationContext UIThread.TheSynchronizationContext

let! decls = req.ResultScope.GetDeclarations(req.View, req.Line, req.Col, req.TokenInfo, reason)

…

}

This code *advertises* that the async in question should run on UI thread.

In general, with the “use! Async.*ThreadAffinity*” the async block can *declare* the thread affinity it requires – it looks lovely!

## ThreadAffinityContext

All of the above is lovely, but unfortunately does not quite work in VS – because SynchronizationContext is such a disaster:

1. As we have learned during dev10 cycle, in VS UI thread does not necessarily has SynchronizationContext.Current set. Therefore even if all other API calls play nicely, we cannot guarantee that they return UI thread after all, so we will still need to litter our code with “do! Async.SwitchToContext UIThread.TheSynchronizationContext” after every “let!” on UI thread – yuck!
2. When GetDeclarations runs its async on reactor thread, it wants every “let!” inside that async to return to reactor thread (that is what “use! Async.ReactorThread” should imply). This means installing SynchronizationContext.Current on reactor thread – but if I do that, I am sure to get an inquisitive e-mail from Doug Hodges or Paul Harrington or such like . In general, setting the current synchronization context is a security critical operation and shouldn’t be required for enforcing async thread affinity.

Generally, the big problem with SynchronizationContext.Current is that it is a universally accessible global and breaks encapsulation (as is universally accessible globals’ habit).

The proposed solution here is to make an equivalent of SynchronizationContext (let’s call it ThreadAffinityContext) a part of Async monadic state (aka AsyncParams).

We will have operations like:

GetThreadAffinity : Async<ThreadAffinityContext>

SetThreadAffinity : ThreadAffinityContext -> Async<unit>

ThreadAffinity : ThreadAffinityContext -> Async<IAsyncDisposable>

The current ThreadAffinityContext of the async monad will be used in places where we today use SynchronizationContext.Current.Post.

For backwards compatibility, the default ThreadAffinityContext asyncs run on will delegate to SynchronizationContext.Current.

We have precedents for the this kind of thing in other async frameworks –TPL and Rx have TaskScheduler and IScheduler abstractions respectively for this kind of stuff. We should check with them for possible alignment here. There are also alignment issues with C#/VB feature that we will need to work out.

## AsyncSeq

Tomas, that’s a wonderful contribution! Indeed, it feels like this may be heading towards something that is worth a paper (all together?).

I’m still mulling over your type definition below. It is certainly beautiful and natural. I would instinctively convert it to the following (following F# lazy lists, and Wadler/Taha’s paper “[How to add laziness to a strict language, without even being odd](http://citeseer.ist.psu.edu/old/725251.html)”, which applies to pretty much an computational modality, not just laziness)

type AsyncSeq<'T> = Async<AsyncSeqCell<'T>>

and AsyncSeqCell<'T> = Nil | Cons of 'T \* AsyncSeq<'T>

with typical operations like these (so simple!!)

let rec map (f: 'T -> 'U) (x: AsyncSeq<'T>) : AsyncSeq<'U> =

    async { let! v = x

            match v with

            | Nil -> return Nil

            | Cons (h,t) -> return Cons(f h, map f t) }

let rec mapAsync (p: 'T -> Async<'U>) (x: AsyncSeq<'T>) : AsyncSeq<'U> =

    async { let! v = x

            match v with

            | Nil -> return Nil

            | Cons (h,t) -> let! h2 = p h in return Cons(h2, mapAsync p t) }

I attach the builder implementation that follows for this type. Everything runs through very smoothly. The tryFinally and tryWith are pretty interesting (I have not written anything like them before), and thus not yet fully tested, though I think they are right, and they pass all the tests I had laid out before.

Now, the question is, **what does this actually do**?

Well, the answer is that it **time is no longer in the picture ,** at leastuntil you start connecting to time-volatileevent streams.  That is, **these are not event streams**. They are simply sequences whose “head/tail” cell is generated asynchronously.

So basically, in every case you get the same answer as a synchronous sequence, e.g. this:

              asyncSeq { for x in asyncSeq { yield 1;

                                             do! Async.Sleep 100

                                             yield 2 } do

                            yield (x,'a')

                            do! Async.Sleep 10

                            yield (x,'b') }

gives

 [(1,'a'); (1, 'b'); (2,'a'); (2,'b') ]

This is regardless of the time delays involved in either sequence – as I said, time is not in the picture. We do these asynchronous waits@

input  Cons(1,tail1)

sub-list-for-1  Cons((1,’a’), tail2)

tail2  <pause-10>  Cons((1,’b’), tail3 )

tail3  Nil

tail1  <pause-100>  Cons(2,tail4)

sub-list-for-2  Cons((2,’a’), tail5)

tail5  <pause-10>  Cons((2,’b’), tail6 )

tail6  Nil

I also like the fact that the relationship to events and observables becomes very clear, e.g. if you look at these, then you see straight away that the asynchronous sequence is continually reconnecting to the event stream and/or observable.

    module AsyncSeq =

        // This may skip event observations if upstream processing takes too long

        let rec ofEvent (ev: IEvent<'T>) : AsyncSeq<'T> =

            asyncSeq { let! v = Async.AwaitEvent ev

                       yield v

                       yield! ofEvent ev }

        // This continually reconnects, and may miss observations if upstream processing takes too long

        let rec ofObservable (ev: System.IObservable<'T>) : AsyncSeq<'T> =

            asyncSeq { let! obs = Async.AwaitObservable ev

                       yield obs

                       yield! ofObservable ev }

And presumably a zip combinator would allow a “merge” story if we wanted it (oh, but “zip” would involve a parallel wait on one of two asyncs – tricky without introducing multi-threading – I hear Dmitry asking for interleaved asyncs on a single thread )

Anyway, all very intriguing, and overall a much cleaner story.

Don

**From:** Tomas Petricek [mailto:tomas@tomasp.net]   
**Sent:** 17 February 2010 21:06  
**To:** Dmitry Lomov; Don Syme; Brian McNamara; Matthew Podwysocki  
**Subject:** RE: AsyncSeq (= Observable) puzzles

I was following the discussion and I can only agree that this is *really* tricky.

**Representing AsyncSeq in F#**

Maybe the problem is that a natural representation of asynchronous sequence in F# is not IObservable, but something like this:

type AsyncSeq<'T> = Nil | Cons of Async<'T \* AsyncSeq<'T>

...the difference would be that this representation is delayed, so when you “bind” on the value of Cons, it evaluates (asynchronously) only one step of the sequence. I believe that this type would naturally work with “async” in the same way F# list works in the context of normal code (you can use it for recursive programming with pattern matching (using “match!” of course!) and the “for” loop would evaluate the asynchronous sequence lazily, so it could behave as the version that doesn’t miss any values, but without any caching.

I haven’t really tried this, so I may be completely wrong. Anyway, if I’m correct, then we could use this data type for asynchronous sequences in F# and then define various conversion functions to get from this to IObservable and back (when getting from IObservable it could use caching or it could miss values). Anyway, this is probably purely theoretical, because adding even more types for asynchronous/reactive/parallel programming would be too confusing.

**AsyncSeq for IObservable**

Back to possible designs of “asyncSeq” computation builder for IObservable:

* I think that it should be resumption (one-shot continuation) and there should be only one program counter  
  (F# asynchronous workflows work this way and I think this an important aspect that makes them intuitive and easy to use)
* I was going to suggest that we add something like AsyncEnumerator (which allows explicit access to elements and implements caching), but it seems that you already have that (AsyncObserver), which is great! The similarity between GetEnumerator and AsyncObserver is probably a great way to explain the programming model.
* For resumption, the only question is whether “for” should automatically cache the values or not. I think losing values when using AwaitEvent is a good thing, but I’m not really sure about “for”.

Are there any other code samples that we would expect to be equivalent? For example this one:

|  |  |
| --- | --- |
| asyncSeq {  **for** *v* **in** *aseq* **do**  *<body>* } | **let** ao = AsyncObserver(*aseq*)  **let rec** loop() = asyncSeq {  **let!** next =ao.Next  **match** next **with**    | Some(v) -> *<body>*; return! loop()  | None -> () |

This would suggest that “for” should be using caching. So, I my vote would probably be to use caching. Another practical question is whether the user can easily choose between the two. For example, can he write **for** v **in** aseq |> AsyncSeq.forgetful **do** if we use the cached option (this seems to be tricky)? If we choose the non-caching version, can the user write **for** v **in** aseq |> AsyncSeq.forgetful **do** (this looks more plausible). Anyway, this may be another thing to consider.

**Off-topic: Asynchronous waiting in while**

And here is one more off-topic note – in the equality above, the recursive version on the right hand side was surprisingly difficult to write! I was expecting something like (with “while!” which does asynchronous waiting when evaluating the condition – technically speaking it takes “unit -> Async<bool>” instead of “unit -> bool”):

**let** ao = *aseq*.GetAsyncEnumerator()

**while!** ao.MoveNext() do

**let** v =ao.Current

*<body>*

Thanks for including me in this discussion!

Tomas

## Making MailboxProcessor MarshalByRef

## Enabling async hops across domains