Active Patterns, 13

DataBinding, 14

Debuggability, 14

Design Issues

Build Systems and Implicit File Dependencies, 3

Communication with Visual Studio, 4

Configuration, 5

Cross Assembly, 5

Generative Types May Emit too Much, 5

Injecting Generative Namespaces, 4

Injection Control, 4

Managing Intermediate Files, 4

Pattern Matching Erased Types, 12

Provider Contract, 5

Provider Isolation, 5

Providers we Ship, 5

Security, 5

Synthetic Extension Properties, 2

Unittestability, 5

Validation of Generated Types, 4

Visibility of Injected Types, 4

EmbedTypesAttribute, 8

Expression Trees, 13, 15

F# Queryable, 5

IExtensibleTypeProvider, 10, 11

IExtensionTypeResolver, 9, 11, 13

Incentives for Provider Writers, 5

Intellisense, 11

Intellisense Suggest, 11

LINQ, 8, 9, 13, 15

Liveness, 10

Multistaging, 15

Non-enumerable Types, 9

Pattern Matching, 12, 13

Plain Old Data, 12

Providers

AOP style façade generator, 15

Compile Environment, 15

Csv, 14

Data interchange type, 14

Dbml, 8, 14

Excel, 10, 14

Flickr, 14

Freebase, 11

FreeBase, 14

GData, 14

IMDB, 14

Java, 8

My, 14

NetFlix, 14

OData, 9, 14

Parser\Lexer Generator, 15

ProgrammableWeb, 14

RDF, 6

Reflecting Internals Exposer, 15

StackOverflow.com, 14

Stocks, 14

Strongly typed .resx generator, 14

Team Foundation Server, 14

US Government Open Datasets, 14

Weather, 14

Wsdl, 7

Xaml, 14

Xml, 14

Quoting, 13

Records, 12

Ruby on Rails, 15

SqlMetal, 15

Terminology, 9

Thin Provider, 8

Type Generation, 8

Type Identity, 10

Visual Studio, 11, 13, 14

Walkthru

Xaml, 3

# Sep 30, 2010 Compiler changes for type providers review

List of issues coming out of the review meeting:

Spec issues:

         What is exact subset of **Type** that ITypeProvider.ResolveTypeName can use?  What happens when the contract is broken?

         What is exact subset of **Expression** that ITypeProvider.GetInvokerExpression can use?  What happens when the contract is broken?

Cross-assembly behavior of type providers:

         How does generative injection-into-user-assembly deal with multiple assemblies (e.g. two different F# projects reference same database, now a third project cannot talk to both those two)? Is generative code always "internal", does that rule out any important scenarios?

* The same for erasure-based providers: (what happens if F# assembly exposes Freebase types on API boundary?)

Type providers and the IDE interaction:

         What data will be carried by TypeProviderConfig? (currently is 'string ResolutionFolder', is that an absolute path? Very hard to be prescriptive here about how TP gets a hold of user’s assets…)

* What is the user interaction in VS? What happens if you program against an OData service and the internet goes down? Does build lab need to be connected to the Internet? What if there are outside changes during build process?

         How user can 'force a reset' of TPs in IDE?  (‘changing the project file’ would e.g. force the LS to invalidate everything, but does it make sense for explicit UI for this?)  What mental model do users need (in general, or on a per-provider basis)?

API issues:

         what does the 'string assemblyName' means on CompilerExtensionAssemblyAttribute constructor (fusion name? file name? ...)

         is text in ExtensionTypeXmlDocAttribute either text or XML blob (how interpreted by IDE?)

         is DefinitionLocationAttribute.FilePath relative or absolute?

         should ITypeProvider.NamespaceName be an array of strings (so no parsing of '.', and to support e.g. ``foo.bar`` names)?

Other things we need to do:

         threat model

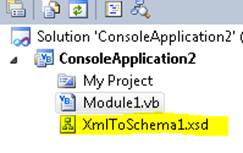
         brainstorming points-of-failure and diagnostics for two kinds of user-scenarios

o   type provider authors

o   users consuming type providers

# May 6, 2010 Rooting EST Types with Extension Properties

VB has an EST-like extension specific to supporting XML directly in the language. The interesting thing from our standpoint is that the type you dot off of is a plain XElement. We could accomplish this effect by supporting extensions properties in EST. The extension type would attach synthetic properties to System.Xml.XElement.





I’m mentioning this because our current solution would be to do something like:

                let x = Data.Xml.``XmlToSchema1.xsd``(@”d:\dd\MQFSharp\_2\src\fsharp\projects\Prototype.FSharp.Compiler\FSharp.Compiler.xml”)

The VB way looks way less surprising to my eyes. There is no mention of a non-standard type, and there is no file-name-in-an-identifier.

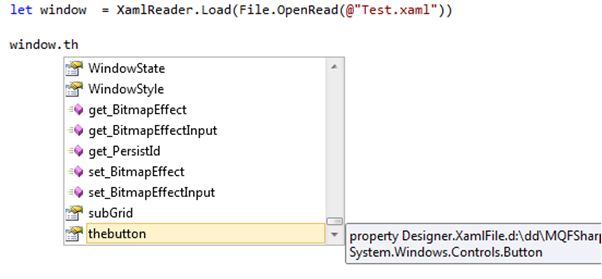
The same principle could apply to any of our file\url based providers: xaml, excel, dbml, odata, etc.

The piece that takes a bit of imagination is this: there could be multiple .xsd files in the project but it doesn’t matter. The first level of dotting shows the combined set of extension properties for all xsds. Once you select an element you have disambiguated which xsd you’re working against.

Recall that one of the messages from the Xaml folks was that this looks weird:

let window = new Designer.XamlFile.``Test.Xaml``()

It could look like this:



# April 20, 2010 Current Xaml Walkthru

This is code used in a recent xaml walkthrough:

#r "Designer.XamlFile.dll"

#r "PresentationFramework"

#r "PresentationCore"

#r "WindowsBase"

#r "System.xaml"

System.IO.Directory.SetCurrentDirectory(\_\_SOURCE\_DIRECTORY\_\_)

let mainWindow = Designer.XamlFile.``MainWindow.xaml``()

mainWindow.Grid.thebutton.Click.Add(fun \_ -> mainWindow.Grid.thebutton.Content<-"Hello")

mainWindow.Visibility<-System.Windows.Visibility.Visible

# April 19, 2010 Build Systems and Implicit File Dependencies

**Summary of the Problem**

EST introduces the possibility that the build may depend on files that fsc.exe and the enclosing build system do not know about.

Consider,

                type Northwind = Data.Sql.``Northwind.dbml``

The correct behavior now is that a change to Northwind.dbml will cause a rebuild. But there is no mechanism for telling a build system about the file, so the rebuild won’t happen.

In traditional code spit systems, this is not a problem because the code generator can respit the Northwind.cs file if the .dbml has changed.

**Partial Solution One – Set the Build Action to ‘CustomAdditionalCompileInputs’**

If the build action for the Northwind.dbml item is set to **CustomAdditionalCompileInputs** then the project will know when to rebuild. There are several ways this build action could get set:

* The provider supplies an item template that sets it when the item is initially added.
* The user sets it manually from within VS.
* The project system ‘intercepts’ add file and rename requests to alter the build action

Problematically, this addresses the problem fairly high in the stack. This means that it will be easy to write a provider that forgets about this issue. The general provider is no longer a nice little xcopy-able entity.

**Partial Solution Two  – A specialized MSBuild Task**

We add a method to each provider:

                member this.AreYouInterrestedInFile(filename:string)

This will return true if the provider is speculatively interested in the file. Now, if the file has changed then a build will be triggered.

An issue here is that it’s not very precise: a provider may be interested in a file that the compile doesn’t ultimately depend on. The function itself seems unrelated to the purpose of the current interface (wrong cohesion).

Also, this fix may still be too high in the stack: it doesn’t help the makefile case.

**Partial Solution Three – fsc.exe reports file dependencies after the build**

We change fsc.exe to emit:

bin\Check\MyAssembly.est-files.txt

This will contain a list of files that the build actually depends on. During the next build, we check those files again.

I think this works, except I’m not sure how to cause make to read and use MyAssembly.est-files.txt. I’m guessing this should be possible though.

# April 13, 2010 Survey of Open Issues

**Communication with Visual Studio** – How can a provider receive notification that a file has changed in memory? Or tell VS to take hands off a file?

**Managing Intermediate Files** – Some providers need to store information locally. For example, intermediate dlls for injection, cached results, etc. What do providers need to manage this well and to live harmoniously in build systems and simple scripts.

**Injecting Generative Namespaces –** What should the design be?

**Visibility of Injected Types –** Should the user be able to control the visibility of generated types? If so how? What is the relationship between visibility of the [<Generate>] type abbreviation and the type it refers to?

**Injection Control --** Putting user more in control on what gets generated. Now it seems too easy to pollute one's assembly with zillion of web services

**Validation of Generated Types** -- needs careful review. Now there are many warnings that do not make sense.

**Configuration –** How to communicate passwords and other configuration information to the providers?

**Providers we Ship –** Figure out which providers, if any, we will ship in Dev11.

**Cross Assembly** -- Generated types visibility, F#-to-F# and F#-to-.Net views.

**Provider Isolation –** How much isolation can\should we provide from compiler\VS to provider? Is exception catching enough? Separate app domain? Separate process?

**Provider Contract –** What errors and warnings do we supply on generative vs. non-generative providers?

**Unittestability –** How can code written against providers that access remote information be unittested in a reliable way? Consider an app written against freebase—can information be stored locally and checked in for freebase to run against?

**Security –** Start with the implications of running user code during compilation.

# April 11, 2010 Generative Types May Emit too Much

From Dmitry in reference to the Dbml provider:

There is certainly too much stuff being generated at the moment, and too easy to generate too much stuff into your assembly. We need to do another round of design meetings on this one.

# April 9, 2010 Overview of EST with FUSE UK

(Don’s notes)

I did an overview of EST with Ralf, Bob, Jurgen, David, Konstantina and Andrew. Great interest all round, and thanks to everyone for coming along. Scripts and slides attached.

* There was a fair bit of discussion about incentivize structure for writing providers. The candidates are
  + Providers of data (but they will often have an existing API and consider that enough)
  + Consumers of data (but quality of providers may suffer)
  + 3rd party individuals, internal teams, companies (good, but quality/cost will be an issue)
  + Microsoft teams
  + etc.

and there are many related factors.  Understanding the incentive structure and tradeoffs will be absolutely essential here in positioning the feature.

* Q: What do we do about many:1, many:many  etc. relations in Freebase?
  + Answer: I don’t know the details, but I need to learn them
  + It’s important people giving talks have a clear idea of exactly what is being done
* Q: a fair bit of discussion about needing to support a query story.
  + Javascript apis have a nice but somewhat adhoc query story (e.g. freebase queries)
  + Developing experience with mapping F# queries into these structures will be really important, and ultimately we either
    1. have to make it simple to implement a reliable strongly-typed query experience (c.f. LINQ) or
    2. make it simple to piggyback off an typed query experience (e.g. if a reliable IQueryable implementation exists)
    3. make it simple to piggyback off an untyped query experience (c.f. ADO.NET)
* Q: We are focusing on importing arbitrary things “into” F#. What about publishing “back”? Should/would a provider help with that?
  + Good question. After all, the fundamental proposition of F# is two-way interop “you can consume any .NET from F#, you can use any F# code you write from .NET”.
  + Argument against: the mappings implemented by the providers are not trivial to invert, e.g. to host your code back as a web service is not trivial – in the case of WSDL, you need a web server, for starters.  And what would this even mean w.r.t. Freebase etc.?
  + Argument against: there are established standards for how to “export” .NET code as, say, a REST API.
  + However, it is true that the techniques to “export” back are far from satisfactory. It would be very very cool if the act of “exporting” F# functionality was somehow as simple simply adding a reference

#r "Web.AutoExportAsREST.dll"

                                Or

#r "Data.AutoExportAsExcelSpreadsheet.dll"

                                Plus some metadata, then running the resulting .EXE 

* + Takeaway: think deeply this question, but no immediate action
* Ralf says we should look at RDF as a schema standard.
* Observation: We will really need to engage F#-positive data experts in this work. FUSE UK may really be able to help us here.
* Observation: these fragments go down very well, especially the ones towards the end (though they also raise questions about the Freebase mapping we’re using) 

Data.Freebase.Chemistry.Isotope |> show

Data.Freebase.Olympics.``Olympic event`` |> Seq.toList |> show

// Task - find all artworks in the database completed between 1850 and 1900

Data.Freebase.``Visual Art``.Artwork |> Seq.toList |> show

[ for work in Data.Freebase.``Visual Art``.Artwork do

     if work.``Date Completed`` > "1850" && work.``Date Completed`` <= "1900" then

        yield work ]

  |> Seq.toList |> show

Data.Freebase.Celebrities.``Abused substance`` |> Seq.toList |> show

Data.Freebase.Celebrities.Celebrity |> Seq.toList |> show

Data.Freebase.Celebrities.``Sexual orientation`` |> Seq.toList |> show

Data.Freebase.Celebrities.Supercouple |> Seq.toList |> show

Data.Freebase.Celebrities.``Romantic relationship`` |> Seq.toList |> show

Data.Freebase.Celebrities.``Rehab facility`` |> Seq.toList |> show

let estimateLikelihoodOfSubstanceAbuse (celebrity: Data.Freebase.DomainTypes.CelebritiesDomain.Types.Celebrity) =

    if celebrity.Name.Contains "Amy" then 95.0 else 15.0

* Observation: The Excel demo raises many hard questions and may not be the best one to start with – Freebase is so compelling (and now very stable) that we should probably just start with that.
  + Using the live set of Excel spreadsheets is kind of cool but a little too dynamic for the first demo.
  + The provider allows you to bake a \_*lot*\_ of static assumptions into your code very easily, e.g.
    1. Excel file name
    2. Cell names
    3. Cell types

This is a bit too controversial for a first demo.

* + I think it is important to be more upfront about which schema choices are being implemented by the Excel provider. This may be as simple as reporting

Data.Excel.Live.ExcelFile.MySpreadsheet.Invoice.RowsByNamedColumn

* + The schemas are implicit in the provider, which is one model, but a somewhat unusual one. That’s good when explaining the underlying F# feature, but starting with a industry standard schema is probably better when selling the language
* Q (Andrew): “how long do the types live”?
  + I went through the details with Andrew after

# April 7, 2010 Programmatically Cracking .wsdl

Information on cracking wsdl (to get comments for example) can be found here:

<http://msdn.microsoft.com/en-us/library/aa717040.aspx>

# April 3, 2010 Java Provider

Thought for the day – what if we had a provider that permitted you to use any Java class file as a .NET component?

namespace SomeJavaStuff = Java.``someComponent.jar``

let thing = SomeJavaStuff.SomeJavaClass()

The invokes would, of course, be via reflection/remoting etc. to a hosted JVM instance. The result would be workable (though not, of course, ideal).

The generative version of this would presumably embed the class file into your app, making deployment easier

[<Generate>]

namespace SomeJavaStuff = Java.``someComponent.jar``

let thing = SomeJavaStuff.SomeJavaClass()

Of course the same would apply for any statically typed or partially-schematized language with sufficient reflection capabilities.

# March 25, 2010 Notes on Erasing OData Provider

The WCF Data Services equivalent of LINQ to SQL’s DataContext is System.Data.Services.Client.DataServiceContext. Interesting to EST is that there are methods on it for type resolution:

        // Summary:

        //     Gets or sets a function to override the default type resolution strategy

        //     used by the client library when you send entities to a data service.

        //

        // Returns:

        //     Returns a string that contains the name of the System.Data.Services.Client.DataServiceContext.

        public Func<Type, string> ResolveName { get; set; }

        // Summary:

        //     Gets or sets a function that is used to override the default type resolution

        //     option that is used by the client library when receiving entities from a

        //     data service.

        //

        // Returns:

        //     A function delegate that identifies an override function that is used to

        //     override the default type resolution option that is used by the client library.

        public Func<string, Type> ResolveType { get; set; }

There appears to be no public API for parsing the $metadata xml but there is a [sample](http://www.odata.org/developers/odata-sdk#/media/7573/odataexplorer.zip) in C# called OData Explorer that parses $metadata with XLinq.

OData has a paging mechanism in which a partial result is returned along with a continuation Uri for the next page.

The equivalent to Table<T> is DataServiceQuery<T> which has a late-bound counterpart DataServiceQuery.

Since OData is a XML\JSon REST API a provider could also choose to plug in at that level.

# March 17, 2010 Type Generation

**High Level Summary**

Consensus was that we will need to be able to generate real types and resources. For types, we think this syntax will be good:

[<EmbedTypes>]

type Northwind = Data.Linq.``connection-string``

This will cause real, and possibly public, types to be emitted into the user’s assembly. Dmitry will prototype this in the Dbml provider and report his results. If a provider requires generation then there will be a red squiggle under **Data.Linq.``connection-string``** with a message that indicates the attribute is required. The name of the attribute was not settled on.

**Cases that Require Real Types**

Reflection over type parameters—like Table<Synthetic>

Thin provider—calls svcutil.exe and just uses those types.

**Argument Against This Feature**

It may be possible to write all the providers we care about with pure erasure.

This seems to bifurcate the user experience. When is [<EmbedTypes>] required? Why?

# March 17, 2010 OData

Luke just pointed this out.

<http://services.odata.org/Northwind/Northwind.svc/$metadata>

<http://services.odata.org/Northwind/Northwind.svc/Products?$format=json>

I think this just went live because I hadn’t seen it before at OData.org. This could form the basis of a nice EST sample.

# March 12, 2010 Terminology

This is the name for two things.

Extensible Static Typing (not Extension Static Typing)

Extension Type (not Extensible Type)

# March 12, 2010 On Demand Types

One way in which EST-synthetized types differ from CLR types is that EST-synthetized types may have on-demand members – members that come into being by virtue of user requesting them. For example, if Data.CsvFile is a synthetic class for CSV file data source, Data.CsvFile.``c:\myCsvFile.csv`` might be a property that returns data from c:\myCsvFile.csv. Likewise for Data.Linq.``<connection string>``. We support this in the compiler by always using GetMember(“name”) on System.Types that EST providers return, and never assuming that GetMembers() return an exhaustive list of members.

However, the need also arises for top-level types to be non-enumerable in the same way. For example, the Linq provider wants to create types named Data.Linq.``<connection string>`` to represent a database behind the <connection string>. So far I have been doing achieving this by having a single synthetic top-level type Data.Linq and manufacturing nested types as needed.

However, for Linq provider I want the types representing databases to be Reflection.Emitted, and Reflection.Emitted types cannot be nested types of other implementations of System.Types.

It also seem more orthogonal to bake the support for on-demand synthetic types into IExtensionTypeResolver interface.

After some discussion, this is the version of the interface Jomo and I have arrived at:

public interface IExtensionTypeResolver

{

    Type[] GetTypes();

    Type GetType(string name, BindingFlags bindingAttr);

    string NamespaceName { get; }

    Expression GetInvokerExpression(MethodBase syntheticMethodBase, ParameterExpression[] parameters);

}

* Every ExtensionTypeResolver contributes types to a single namespace (NamespaceName)
* Compiler queries ETR for a particular type via GetType method (bindingAttr parameter is there primarily to support case-insensitive lookup)
* GetTypes() is used to provide IntelliSense

# March 10, 2010 Type Identity

If you compile two assemblies with the same SQL Customer type in them then are they assignable to each other? Can they be shared across public boundaries?

It seems like there are two possible approaches that would let us have a consistent story (I don’t know yet if these are actually possible).

**Provider-based Identity**

In this world, EST types look and act as if they are public types in the provider assembly. These types need to look as if they are assignable to each other. They will be versioned by the strong identity of the provider assembly. I’m not sure this is possible in a cross-language way. NoPIA solves this problem in a NoPIA-specific way.

**User assembly-based Identity**

EST types look and act as if they are internal types in the user’s  assembly. We would not allow these ESTs to declare themselves public. These types are not assignable to each other across assembly boundaries. Versioning doesn’t matter here.

This is an open design topic and I will put this in the design notes.

# March 10, 2010 Liveness

Dmitry and I had an office conversation about the API changes necessary to enable the Excel liveness scenario. In this scenario, we would like to change something in Excel and immediately see those changes reflected in Visual Studio. We discussed several options for the API and finally settled on this proposal:

public delegate void TypeChangedEventHandler(object sender);

public interface IExtensibleTypeProvider

{

    event TypeChangedEventHandler TypeChanged;

    Type[] TopLevelTypes { get; }

    Expression GetInvokerExpression(MethodBase syntheticMethodBase, ParameterExpression[] parameters);

}

The event will fire if any exposed type’s underlying representation has changed. When the event fires, all types from the given resolver should be considered invalid by the language service.

**Lifetime of IExtensibleTypeProvider**

The IExtensibleTypeProvider will be instantiated when the build references the assembly that contains it. It will be destroyed and recreated after it fires the TypeChanged event

**Scope of IExtensibleTypeProvider**

In VS there will be a separate instance of IExtensibleTypeProvider for each project that references the assembly that contains it.

**Other Possible Designs**

We considered several other possibilities.

**IObservable** This is not available in 3.5 and it doesn’t seem like a big enough issue to consider stopping 3.5 support just for this.

    public class ExtensionTypeChangeMessage {}

    public interface IExtensibleTypeProvider

    {

        IObservable<ExtensionTypeChangeMessage> Observe();

**TimeStamp Attribute** We’d put a synthetic time stamp attribute on the top level synthetic types. The incremental builder would poll the types by calling GetCustomAttributes and rebuild when a type has changed.

        [<TimeStamp(152)>]

        type SyntheticType = class end

This seems too indirect. Almost any good provider is going to need to think about this issue so it seems more appropriate for this to be an interface method. Also, a polling model may be have somewhat worse performance because the change may happen at the very beginning of the polling interval.

**TimeStamp Property** This would be more direct than using the attribute. It still suffers from the polling weakness. Also, the operation may be time consuming and this would block the background thread.

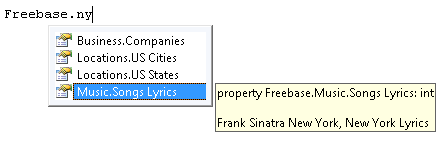
    public interface IExtensionTypeResolver

    {

        int TimeStamp { get; }

# March 10, 2010 Intellisense Suggest

Luke originally mentioned this idea and I wanted to share it out more broadly. In particular with services like Freebase there are a huge number of types available (about 2000 right now and growing). Luke suggested the idea that the Intellisense dot metaphor could be extended to work more like Suggest in Bing. Here’s a dummied-up screenshot:



Instead of just navigating one level at a time and by name, we could suggest completions that are several levels deep and base the suggestions on the content of the underlying store (rows in a database, XML comments, etc).

This isn’t a fully-formed design but it seems to me that there is real potential here and that it could mesh nicely with the “explorative” nature of F#.

# March 3, 2010 Pattern Matching Instances of Erased Types

I think it’s safe to say that 90% of the usefulness of ESTs is for getting at data. I really think there needs to be a way to efficiently pattern match them since that’s the way you deal with data in F#. (From here on, I’m not talking about inferring EST types. I just want to pattern match them).

**Idea 1 – Allow ESTs to explicitly be records**

In this world, we annotate ESTs in a way that allows them to be treated as records by our compiler. For example, the EST could look like:

[TreatAsRecord]

class MyRecord

{

    public string MyProperty {get;}

}

At compile time, F# would validate that the type meets “recordness” criteria—must be sealed, all public fields, no interface implementations, whatever. From there, we allow the type to go anwhere a record can go in F#. Type inference would fall out of this but I don‘t really care about that except to the extent that we’re consistent with true record behavior.

Dmitry: This feels like a slippery slope towards “any synthetic types”. If records, why not algebraic data types? Units-of-measure?

I really like the idea of EST providers only ever synthetizing CLR-like types, and then making sure that any language features we add to benefit consuming EST types also benefit consuming regular CLR types.

Besides, this will make ESP providers more widely applicable - if EST provider synthetize F# record it cannot be used in C#.

**Idea 2 – Define ‘Plain Old Data’ types and allow pattern matching over it**

Let’s define plain old data (POD) types as basic CLR values types, classes with only public fields of PODs, structs with public fields of PODs, lists of PODs and arrays of PODs.

Then we extend pattern matching to work with PODs using an added syntax.

match excelRow with

| {- Quantity = 25 -} -> // Do something

| {- Description = x -} -> // Do something with x

Don: I’m not a big fan of record patterns, one reason why they are not often known.

Sitting in an airport lounge is not the right time to discuss the pros and cons, but the natural direction for the language would probably be more towards active patterns, e.g.

* first address the current limitation and allow static and instance members to be active patterns
* then allow .NET assemblies to define active patterns (by simply having static/instance members with the right name)
* then allow named pattern arguments to active patterns (Dmitry is right that this is an issue)
* then allow EST active patterns

The first three steps are prerequisites and they would compete with lots of other possible design extensions/additions/cleanup

Or simply not supporting extensible pattern matching via ESTs on a first version.

Continuing to collect motivating scenarios would be good.

# March 1, 2010

Discussion of Expression Trees.

* + It feels like restricting to a method call is not the right thing (at least allow a constant expression itself).
  + It feels like we should allow a “Expression.New”, a property get, property set etc. (the later can be currently simulated well enough with get\_ and set\_ calls, but for the purposes of quotations it would be important to support them) The minimal stable position here feels like you should be able to use the mechanism to “forward” a System.Type object model to an isomorphic alternative implementation of the object model. We would be very likely to make use of that, e.g. for LINQ support.
  + What if the method call is to a special F# compiler intrinsic in FSharp.Core (e.g. addition). It feels like we should either exclude those or map those back to uses of the appropriate precise TcGlobals vref at the appropriate instantiation.

# February 26, 2010

Collected notes and design topics from various discussions

* Pattern matching on extension types
* typeof<ExtensionType>, extensionInstance.GetType()
* What sorts of type are allowed? (Must be sealed, no interfaces, members must be public, etc)
* Getting parameters like passwords and urls and connection strings in.
* Structurally named types
* Change propagation into Visual Studio
* Review of IExtensionTypeResolver
* Can we handle the case in which runtime type is required? (SqlMetal)
* Rules for namespace conflict resolution
* Project to Project boundary. Does it work
* Quoting extension type calls
* Getting .resources from provider into compiled image
* Debuggability of extension hosts: what kinds of logging?
* Visual Studio challenges of tiny parsable identifier pattern

# February 2, 2010

Notes on types of EST providers

**Data**

Flickr

Stocks

Excel (We have a prototype)

Xml

Dbml

IMDB

Weather

FreeBase (We have a prototype, but there is plenty to do here)

CSV (I have this started but it is not working right now)

NetFlix

Team Foundation Server

StackOverflow.com

US Government Open Datasets

Strongly typed .resx generator (like Brian’s tool but without the tool)

OData– Open Data Protocol

GData

My (like VB’s)

http://www.programmableweb.com/apis/directory/1?sort=category

**Non Data**

Xaml

Data interchange type This provider would define a common data interchange type system that each of the data providers could use to return results. It would include support properties for the basic types (int,string,etc) and other data interchange types. It would also support lazy. The type should implement the interfaces necessary to support databinding (like to Xaml table or tree). Naming would be like:

FSharp.Data.Structural.``{x:int;y:int}``

AOP style façade generator This provider would let you wrap an existing type and then add before and after handlers as well as interception. Like,

type InterceptedString = FSharp.Data.Intercept<System.String>()

let s = InterceptedString(“Hello”)

InterceptedString.BeforeToLower.Add(fun s->printfn “About to call ToLower on %s” s)

s.ToLower()

After compilation the type will have erased to System.String but the event calls will be left in place.

Reflecting Internals Exposer. This provider would let you unittest internal functions without using InternalsVisibleTo:

type ExposedEntityFlags = FSharp.Data.ExposedInternals.``Microsoft.FSharp.Compiler.Tast.EntityFlags``

let ef = ExpostedEntityFlags()

ef.IsPrefixDisplay

Compile Environment

let builtBy = CompileEnvironment.UserName

let builtOn = CompileEnvironment.MachineName

let thisLine = CompileEnvironment.ThisLine

let thisFile = CompileEnvironment.ThisFile

let thisFile = CompileEnvironment.ThisMethod

With Joe’s expression tree changes we could bake in the literal string “Redmond\SomeUser”, “MyMachine”, etc

Parser\Lexer Generator

Equivalent of NoPIA

# July 15, 2008 Multistaging

Just a minor note. Luke, Jomo and I have this on-going discussion about whether data access should be multi-stage (e.g. with code generation, like LINQ) or whether it is better if it is “more dynamic”, like Ruby on Rails.

I just got to a point in some testing work where I needed to run SQLMetal to generate a OR mapping. Then I had to run CSC.EXE, then I had to edit project properties to reference the generated DLL.

I would so dearly, dearly have loved to be able to simply do something like this:

        #r "SqlMetal.exe NORTHWND.MDF"

Or just

        #r "NORTHWND.MDF"

And keep on going in FSI.  I guess my philosophy is “we know how to turn lots of entities into strongly-typed DLLs under standard settings. Why can’t I just #r the entity and be done with it?”

Looking at this I get the feeling it would be such an amazing, killer feature.

Don