F# 3.0 Type Providers

This document describes the behavior of F# 3.0 type providers and their interaction with hosting compilers.

Table of Contents

[1. Introduction 4](#_Toc300653100)

[2. Scenarios, Design Overview 4](#_Toc300653101)

[2.1 Design Overview 5](#_Toc300653102)

[2.2 Specific Scenarios 6](#_Toc300653103)

[3. Logical Characteristics of Type Providers and Provided Types 6](#_Toc300653104)

[3.1 Generated v. Erased Types 6](#_Toc300653105)

[3.2 Erasure of Types 7](#_Toc300653106)

[3.3 Characteristics of provided type definitions 7](#_Toc300653107)

[3.3.1 Assembly and Context 7](#_Toc300653108)

[3.3.2 Kind 8](#_Toc300653109)

[3.3.3 Inheritance 8](#_Toc300653110)

[3.3.4 Members 9](#_Toc300653111)

[3.3.5 Attributes 10](#_Toc300653112)

[3.3.6 Accessibility 10](#_Toc300653113)

[3.3.7 Generics 10](#_Toc300653114)

[3.3.8 C#-isms 11](#_Toc300653115)

[3.4 Type Equivalence 11](#_Toc300653116)

[4. Additions to F# Syntax 11](#_Toc300653117)

[5. Additions to F# Checking Rules 11](#_Toc300653118)

[5.1.1 Inheritance Conditions 11](#_Toc300653119)

[5.1.2 Runtime type tests against erased types 12](#_Toc300653120)

[5.1.3 Checks on Type Definitions 13](#_Toc300653121)

[5.2 Static Parameters 13](#_Toc300653122)

[5.3 Actual Static Parameters 14](#_Toc300653123)

[5.4 Quotations 14](#_Toc300653124)

[6. Detailed API Descriptions (Microsoft.FSharp.Core.CompilerServices) 15](#_Toc300653125)

[6.1 TypeProviderConfig class 15](#_Toc300653126)

[6.2 IProvidedNamespace 16](#_Toc300653127)

[6.2.1 IProvidedNamespace.NamespaceName 16](#_Toc300653128)

[6.2.2 IProvidedNamespace.ResolveTypeName 16](#_Toc300653129)

[6.2.3 IProvidedNamespace.GetNestedNamespaces 16](#_Toc300653130)

[6.2.4 IProvidedNamespace.GetTypes() 16](#_Toc300653131)

[6.3 ITypeProvider 17](#_Toc300653132)

[6.3.1 ITypeProvider.GetNamespaces 17](#_Toc300653133)

[6.3.2 ITypeProvider.GetStaticParameters 17](#_Toc300653134)

[6.3.3 ITypeProvider.ApplyStaticArguments 17](#_Toc300653135)

[6.3.4 ITypeProvider.GetInvokerExpression() 18](#_Toc300653136)

[6.3.5 ITypeProvider.Invalidate 18](#_Toc300653137)

[6.4 Special Attributes 19](#_Toc300653138)

[6.4.1 TypeProviderXmlDocAttribute 19](#_Toc300653139)

[6.4.2 TypeProviderDefinitionLocationAttribute 19](#_Toc300653140)

[6.4.3 TypeProviderEditorHideMethodsAttribute 19](#_Toc300653141)

[6.4.4 TypeProviderAttribute 20](#_Toc300653142)

[6.4.5 TypeProviderAssemblyAttribute 20](#_Toc300653143)

[6.4.6 GenerateAttribute (Microsoft.FSharp.Core) 21](#_Toc300653144)

[7. Type Provider Assemblies and Instances 21](#_Toc300653145)

[7.1 Type Provider Design-time Assemblies 21](#_Toc300653146)

[7.2 Type Provider Runtime Assemblies 22](#_Toc300653147)

[7.3 Hosting Type Providers: Creation, Lifetime and Disposal 22](#_Toc300653148)

[7.4 Hosting Type Providers: Threading 23](#_Toc300653149)

[8. Provided Elements 23](#_Toc300653150)

[8.1 Provided Generated Assemblies 23](#_Toc300653151)

[8.1.1 Required Characteristics 23](#_Toc300653152)

[8.2 Provided Type Definitions 23](#_Toc300653153)

[8.2.1 Required Characteristics of Provided System.Type Definitions 23](#_Toc300653154)

[8.3 Provided Types 25](#_Toc300653155)

[8.3.1 Required Characteristics 25](#_Toc300653156)

[8.4 Provided Methods 26](#_Toc300653157)

[8.4.1 Required Characteristics 26](#_Toc300653158)

[8.5 Provided Properties 26](#_Toc300653159)

[8.5.1 Required Characteristics 26](#_Toc300653160)

[8.6 Provided FieldInfos 27](#_Toc300653161)

[8.6.1 Required Characteristics 27](#_Toc300653162)

[8.7 Provided EventInfos 27](#_Toc300653163)

[8.7.1 Required Characteristics of Design-time Provided System.EventInfo’s 27](#_Toc300653164)

[8.8 Provided Static Parameters 28](#_Toc300653165)

[8.8.1 Required Characteristics 28](#_Toc300653166)

[8.9 Provided Parameters 28](#_Toc300653167)

[8.9.1 Required Characteristics of Provided ParameterInfos 28](#_Toc300653168)

[8.10 Provided Measure Annotations 29](#_Toc300653169)

[8.10.1 Examples Implementing System.Type Manually 29](#_Toc300653170)

[9. Generated Code 31](#_Toc300653171)

[9.1 Static Linking of Generated Assemblies 31](#_Toc300653172)

[9.2 Dependencies of Generated Code 31](#_Toc300653173)

[9.3 Support Embedding Resources 32](#_Toc300653174)

[10. Exceptions and Diagnostics from Type Providers 32](#_Toc300653175)

[11. Cross Assembly References 32](#_Toc300653176)

[11.1 Cross Assembly References for Erased Types 32](#_Toc300653177)

[11.2 Cross Assembly References for Generated Types 32](#_Toc300653178)

[12. Intellisense, Quick Info, Method Tips 32](#_Toc300653179)

[12.1 QuickInfo 32](#_Toc300653180)

[13. Deploying Type Providers 33](#_Toc300653181)

[13.1 Design-time Loader Resolution for Type Provider DLLs and Dependencies 33](#_Toc300653182)

[13.2 xcopy installation 33](#_Toc300653183)

[13.3 Installation into AssemblyFolders 34](#_Toc300653184)

[13.4 Referencing Type Providers 34](#_Toc300653185)

[14. Security and Threat Modeling 34](#_Toc300653186)

[15. Debugging and Profiling Type Providers 35](#_Toc300653187)

[16. Type Provider Emit API (Out of Band Deliverable) 35](#_Toc300653188)

[16.1 An Example 35](#_Toc300653189)

[16.2 Details 35](#_Toc300653190)

[16.2.1 Providing Type Definitions 35](#_Toc300653191)

[16.2.2 Providing Nested Type Definitions 35](#_Toc300653192)

[16.2.3 Providing Properties 35](#_Toc300653193)

[16.2.4 Providing Methods 35](#_Toc300653194)

[16.2.5 Providing Type Definitions with Static Parameters 35](#_Toc300653195)

[16.2.6 Providing Measures 35](#_Toc300653196)

[17. User Education Material (Out of Band Deliverable) 36](#_Toc300653197)

[17.1 Samples 36](#_Toc300653198)

[17.2 Hands-On-Lab: Writing an Erasing Type Provider 36](#_Toc300653199)

[17.3 Hands-On-Lab: Writing a Generative Type Provider 36](#_Toc300653200)

[18. Cut Features 36](#_Toc300653201)

[18.1 Type Provider Isolation & Unloading 36](#_Toc300653202)

[18.2 Managing Build Dependencies and Files 36](#_Toc300653203)

[18.3 Watson Dumps sent to Type Provider Authors 37](#_Toc300653204)

[18.4 Supported SDK for writing type providers 37](#_Toc300653205)

[18.5 Binary Compatibility and Type Providers 37](#_Toc300653206)

[18.6 ResolveExtensionReference 37](#_Toc300653207)

[18.7 Architecting a Live Connection to Visual Studio Buffers 37](#_Toc300653208)

[18.8 F#-isms 38](#_Toc300653209)

[19. Risk Mitigation 38](#_Toc300653210)

# Introduction

This spec covers the specifics of the type provider interface and the responsibilities of the type provider author, the hosting compiler and the language service.

This is a technical document rather than a gentle introduction or how-to guide.

# Scenarios, Design Overview

The purpose of the F# 3.0 Type Providers feature is to take external structured information such as database tables, Excel documents, xml, web services, and to project it into the .NET type system so that it can be programmed in a natural way. See the companion F# Type Providers White Paper for a motivational overview.

## Design Overview

**Cooperative.** Type Providers are designed to require the cooperation of each participating compiler or host tool. The cooperation of the language service for the particular language is also required. Type providers can, in theory, be hosted in tools other than compilers.

**Completeness.** Type Providers project information into a subset of the .NET type system. The elements included in this subset are described in this document. The guiding principles used for deciding these elements are as follows:

* Our guiding principle is: **Type providers can provide types in C# 1.x type system, with the extension that those classes can reference instantiations of generic types anywhere type definitions can reference types.**
  + We explicitly rule out or ignore C# 3+-isms such as extension members, dynamic, etc.
  + Provided types can implement IQueryable<\_>
  + Provided types can implement methods with byref arguments
  + That said, some additional constraints do apply, w.r.t. fields.
* Notes: We aim to avoid approaching the .Net type system on a piecemeal basis. A type provider author should be able to easily assess whether his code generator (be it for data, web service, or any other purpose) can be converted to a type provider. It shouldn’t be that the user is few weeks into writing a type provider and discovers that the technology just does not work for her.

**Scaling.** Type Providers are designed to scale to enormous universes of types, such as those reported by ontology providers, directory providers, or provided by typed projections of CRM or Sharepoint-based information.

**Optional Erasure.** Type providers are designed to allow design-time-hosting of either “erased” provided types, or the compile-time injection of “generated” provided types. This includes the types provided for ontology mappings such as Freebase or WMI. The introduction of erased types requires the introduction of a number of new language rules, described in this document. Most of these can be analyzed through the lens of “what if F# allowed the definition of erased types directly in the language”.

**Optional Hidden Code Generation.** Type providers are designed to allow design-time-hosting and compile/runtime-injection of “real” provided .NET types, including the types generated by tools such SQLMETAL.EXE, DATASVCUTIL.EXE and SVCUTIL.EXE. The use of generated .NET types is designed to be “hidden” from the user with the exception of the use of an attribute to indicate that generation occurs into a particular .NET component in component compilation scenarios.

**Static Parameterization.** Provided types can accept static parameters such as URLs (for services), file names (for data files or service specifications), flags (to control the presented API) and other literal constants. Among other things, this enables providers to implement value-dependent meta-programming.

**Cross-targeting.** Although type providers are .NET 4.0 components, they can provide types relevant to compilation on Silverlight, .NET 2.0, CompactFramework and Phone. This is done by having the provide use a ReflectionOnlyLoad on the relevant related runtime assemblies.

**Integration with Tooling.** Type providers are designed to allow integration into a tooling environment, in particular (though not limited to) Visual Studio. This is indicated through the presence of a number of design features, such as provider.GetTypes (for intellisense completion), provider.Invalidate (for reactions to schema changes that invalidate provided types) and XML documentation for provided methods and properties.

## Specific Scenarios

The technology must be adequate to cover the following specific scenarios:

* Freebase (+ other ontologies)
* Microsoft Windows Azure Marketplace DataMarket
* WMI (non-generative)
* OData (datasvcutil.exe)
* WSDL (svcutil.exe)
* XML/XSD (xsd.exe)
* LINQ-to-SQL (generative, via sqlmetal.exe)
* LINQ-to-Entities (generative, via edmgen.exe)
* Resources (generative, via resgen.exe)
* XAML (non-generative)
* CRM (non-generative)
* LINQ-to-SQL (non-generative)
* LINQ-to-Entities (non-generative)
* WMI (calling mgmtclassgen.exe)

# Logical Characteristics of Type Providers and Provided Types

## Generated v. Erased Types

Each provided type may be “generated” or “erased”

* If it has a type definition under a path D.E.F, and the .Assembly of that type is in a different assembly A to the provider’s assembly, then that type definition is a “generated” type definition. Otherwise it is an erased type definition.
* Generated type definitions are declared as follows:

[<Generate>]

type *TypeName* = *TypeGenerator*<*static-parameters*>

This forces the user of a generative provider to indicate the point-of-generation of the types. Types embedded via a “Generate” definition can be public or internal:

[<Generate>]

type Bing = Samples.WebDirectory.ProgrammableWeb.Bing

or

[<Generate>]

type internal Bing = Samples.WebDirectory.ProgrammableWeb.Bing

* Erased type definitions must return TypeAttributes with the IsErased flag set, value 0x40000000 and given by the F# literal TypeProviderTypeAttributes.IsErased.
* Generated type definitions must only be returned as nested types of an erased type definition.

In the above example, Samples.WebDirectory.ProgrammableWeb is an erased “container” type definition.

This restriction prevents “opening” a namespace which contains generated type definitions.

* When a provided type definition is generated, its whole assembly A is treated as an injected assembly which is statically linked into the resulting assembly.
  + An error should be given if Generate is not used

## Erasure of Types

The types reported by type providers may be *erased*. In this case, the types and method calls are removed entirely during compilation and replaced with other representations. When an erased type is used, the compiler will replace it with the first concrete type in its inheritance chain as returned by type.BaseType.

This means the erasure function we are using is “erasure(x) = the-nearest-non-erased-base-type-of(x)”.

The erasure of an erased interface type is “object” (object is the nearest concrete base type). This erasure function satisfies these properties:

* Preserves interface implementation of concrete interface types
* Preserves concrete class hierarchy

Erasure is from the F# static type system to the IL static type system, which is still different from the runtime types.

## Characteristics of provided type definitions

### Assembly and Context

Provided type definitions may report constituent types from the appropriate load context

For example, if a type definition reference is for targeting Silverlight, then all the “bool”, “int” types etc. must be drawn from the mscorlib.dll assembly for that target platform. This must be implemented using ReflectionOnlyLoad context.

### Kind

Provided type definitions may be classes

This includes both erased and concrete types. This corresponds to the type.IsClass property returning true.

Provided type definitions may be interfaces

This includes both erased and concrete types. This corresponds to the type.IsInterface property returning true. Only one of IsInterface, IsClass, IsStruct, IsEnum, IsDelegate, IsArray may return true.

Provided type definitions may be static classes

This includes both erased and concrete types.

Provided type definitions may be abstract

Provided type definitions may be sealed

Note that sealed erased types may get unsealed representation at the IL level.

Generated provided type definitions may be enumerations

Erased type definitions may not be enumerations

Generated provided type definitions may be structs

Erased type definitions may not be structs

Generated provided type definitions may be delegates

Erased type definitions may not be delegates

Provided type definitions may not be arrays

This means the type.IsArray property must always return false. Provided types used in return types and argument positions may be array “symbol” types, see below.

### Inheritance

Provided type definitions may report base types

See “erasure” for some restrictions here.

Provided type definition may report interfaces

See “erasure” for some restrictions here, e.g.

* Erased class types may not report concrete interfaces
* Erased interface types may report concrete interfaces

### Members

Provided type definitions may report methods

This corresponds to non-null results from the type.GetMethod and type.GetMethods of the provided type definition. The results returned by these methods must be consistent.

* Provided methods may be static, instance and abstract
* Provided methods may not be class constructors (.cctor). By .NET rules these would have to be private anyway.
* Provided methods may be operators such as op\_Addition.

Provided type definitions may report properties

This corresponds to non-null results from the type.GetProperty and type.GetProperties of the provided type definition. The results returned by these methods must be consistent.

* Provided properties may be static or instance
* Provided properties may be indexers. This corresponds to reporting methods with name Item, or as identified by DefaultMemberAttribute non-null results from the type.GetEvent and type.GetEvents of the provided type definition. The results returned by these methods must be consistent. This include 1D, 2D, 3D and 4D indexer access notation in F# (corresponding to different numbers of parameters to the indexer property).

Provided type definitions may report constructors

This corresponds to non-null results from the type.GetConstructor and type.GetConstructors of the provided type definition. The results returned by these methods must be consistent.

Provided type definitions may report events

This corresponds to non-null results from the type.GetEvent and type.GetEvents of the provided type definition. The results returned by these methods must be consistent.

Provided type definitions may report nested types

This corresponds to non-null results from the type.GetNestedType and type.GetNestedTypes of the provided type definition. The results returned by these methods must be consistent.

The nested types of an erased type may be generated types in a generated assembly. The type.DeclaringType property of the nested type need not report the erased type.

Provided type definitions may report literal (constant) fields

This corresponds to non-null results from the type.GetField and type.GetFields of the provided type definition, and is related to the fact that provided types may be enumerations. The results returned by these methods must be consistent.

Provided type definitions may NOT report non-literal (i.e. non-const) fields

This is a deliberate feature limitation, because in .NET, non-literal fields should not appear in public API surface area.

Note: this could theoretically be lifted for concrete provided types. It would meet our “.NET 1.x” design criteria.

This could also be lifted for erased types. In both cases this would currently be at the cost of changing the ITypeProvider interface, and in this case properties can suffice as a replacement. Decision: No gain in supporting non-literal fields in erased types.

### Attributes

Provided type definitions, properties, constructors, events and methods may report attributes

This includes ObsoleteAttribute and ParamArrayAttribute attributes

### Accessibility

All erased provided type definitions must be public

However, concrete provided types are each in an assembly A that gets statically linked into the resulting F# component. These assemblies may contain private types and methods. These types are not directly “provided” types, since they are not returned to the compiler by the API, but they are part of the closure of the types that are being embedded.

### Generics

Provided type definitions may not be generic

Provided method definitions may not be generic

This is a deliberate feature limitation to reduce the scope of the feature appropriately.

This corresponds to

* GetGenericArguments returning length 0
* For type definitions, IsGenericType returning false
* For type definitions, IsGenericTypeDefinition returning false
* For method definitions, IsGenericMethod returning false
* For method definitions, IsGenericMethodDefinition returning false
* Otherwise a TPV error is reported.

### C#-isms

If a provided type definition reports an extension member, it is not treated as an extension member

Extension members are reported through ExtensionMemberAttribute. These are ignored.

## Type Equivalence

Two generated provided type definitions are equivalent if and only if they have the same F# path and name in the same assembly, once they are rooted according to their corresponding generative type definition.

Two erased provided type definitions are only equivalent if they are provided by the same provider, using the same type name (for ResolveTypeName), with the same static arguments (for ApplyStaticParameters).

# Additions to F# Syntax

The syntax of types in F# is expanded to include static parameters, including named static parameters:

typeArgValue =

| const expr

| sbyte | int16 | int32 | int64 | byte | uint16

| uint32 | uint64 | bool | char | string | decimal | single | double

typeArg +=

| typeArgValue

| id '=' typeArgValue

static-arguments =

<typeArg,…,typeArg>

This means references to provided types may include static parameters, e.g.

type SomeService = ODataService<"http://some.url.org/service">

Parentheses are needed around any expressions after “const” that are not identifiers, e.g.

type K = N.T< const (+1) >

# Additions to F# Checking Rules

### Inheritance Conditions

Concrete type definitions (both provided and F#-authored) may not inherit from erased types

Concrete type definitions (both provided and F#-authored) may not implement erased interfaces

Erased type definitions may not implement new concrete interfaces

In .NET, listing an interface implementation on a type implies giving an actual new implementation and that that implementation replaces all existing implementations. For an interface implementation on an erased type, this new implementation would not replace existing implementations from the .NET perspective, so it would feel incorrect to permit it.

A type (or object expression) may implement IConcrete<ErasedType>

A type (or object expression) may not implement IErasedType

Interface sets must be unique subject to erasure

### Runtime type tests against erased types

Type tests against nominal erased types

For example:

   x :? IErasedType

In this situation, an error must be reported.

We do need to give an error for direct type tests against erased types E, and a warning for type tests against non-erased types whose instantiation include erased types, e.g. List<E>. Note the latter is the same issue as UoM type tests - we punted giving warnings for these from dev11– it is time we addressed both.

**Type tests against partially erased nominal types**

For example:

   x :? List<IErasedType>

In this situation, a warning must be reported.

**Type tests against naked generic type variables**

For example:

let testAlpha<'a> (x:System.Object) =

    match x with

    | :? 'a as x2 -> sprintf "have %A" x2

    | \_           -> "not"

testAlpha<T2> (T2())

No warning is given.

Note: We don’t need to give warnings for type tests involving generic parameters like the one below. Logically speaking, a warning in this situation makes sense: the type analysis is incomplete. As such, I don’t mind an optional level 4 warning, but we don’t want one on by default, because there’s nothing that the user can actually do to act on the warning.

Note: We could have a “T : concrete” constraint that indicates a type variable has been used for type tests. That is under consideration but is orthogonal to the type providers feature.

### Checks on Type Definitions

Methods must be unique subject to erasure

For example, the following must give an error:

type T() =

    method this.Foo(x : Freebase.``Chemical Element``) = ..

    method this.Foo(x : Freebase.Celebrity) = ....

because it erases to

type T()

    method this.Foo(x : FreebaseObject) = ...

    method this.Foo(x : FreebaseObject) = ...

An error must be given here.

## Static Parameters

During type checking of a type A<static-arguments>opt, where A is a provided type, the compiler calls GetStaticParameters to determine the static parameters for the type A. If the provided static parameters match, it then calls ApplyStaticArguments to apply the static arguments to the (unapplied) provided type A. This returned type is then queried like any other System.Type.

Each static parameter must report one of the following types via the param.ParameterType property:

* byte, int16, int32, int64, sbyte, uint16, uint32, uint64, decimal, single, double, char, string, bool.
* Any enum with an underlying type that is one of the above types

Unlike custom attribute parameters, static parameters may not be arrays, and may not have ParameterType “System.Type”. A string giving a full name for the System.Type may, in some circumstances, be used instead.

Static parameters may be optional and/or named, indicated by the Attributes property of the ParameterInfo.

The provider must return a System.Type with Name **typeNameAfterArguments**. This name will include an encoded form of the static arguments to allow later relinking of references to this type in F# metadata. The encoding scheme used is

**encoding**(A<arg1,…,argN>) =

typeName,ParamName1=**encoding**(arg1),…, ParamNameN=**encoding**(argN)

**encoding**(*v*) = "*s*"

where *s* is the result applying the F# ‘string’ operator to v (using invariant numeric formatting), and in the result " is replaced by \" and \ by \\

Named static arguments must come after all other arguments.

Relevant errors for named and optional static parameters are:

No static parameter exists with name '%s'

Multiple static parameters exist with name '%s'

Named static arguments must come after all unnamed static arguments

The static parameter '%s' requires a value

## Actual Static Parameters

The syntax of types Name<type,…,type> is interpreted as an application of a statically parameterized provided type if Name is a provided type with static parameters.

The actual static arguments may be any of:

* Constants of the appropriate type
* Literals
* **const** expr, where expr is a constant expression that is a valid argument to a custom attribute.

The syntax of F# types is extended with

* type := … | **const** atomicExpr

The expression syntax here is only constants, identifiers, or parenthesized expressions.

Note: In F# 2.0/3.0, the only non-trivial interesting custom attribute expressions are bitwise-or operations on integer and enumeration values.

## Quotations

Type checking of quotations is w.r.t. unerased type checking rules. Runtime values of quotation literals are w.r.t. erased representations. This is the same as units of measure today.

Note: we must test quotations of provided types as rigorously as we test codegen fo r provided types.

# Detailed API Descriptions (Microsoft.FSharp.Core.CompilerServices)

The type provider author must implement the ITypeProvider interface.

public interface IProvidedNamespace

{

string NamespaceName { get; }

IProvidedNamespace[] GetNestedNamespaces();

Type[] GetTypes();

Type ResolveTypeName(string typeName);

}

public interface ITypeProvider

{

IProvidedNamespace[] GetNamespaces();

Assembly ResolveExtensionReference(string name);

ParameterInfo[] GetStaticParameters(Type typeBeforeArguments);

ParameterInfo[] ApplyStaticArguments(Type typeBeforeArguments,

string typeNameAfterArguments,

object[] staticArguments);

Expression GetInvokerExpression(MethodBase syntheticMethodBase,

ParameterExpression[] parameters);

event System.EventHandler Invalidate;

}

## TypeProviderConfig class

Type providers are configured by the compiler using an instance of the following type supplied as an argument to the constructor of the type identified by TypeProviderAttribute. If no constructor exists taking an argument of this type, a parameterless constructor must be present and is used instead.

sealed public class TypeProviderConfig

{

public TypeProviderConfig(string resolutionFolder);

/// The folder in which the resolution is occurring.

/// Typically the project or script folder

public string ResolutionFolder { get; }

/// Full path to actual referenced assembly that caused

/// this type provider to load and instantiate

public string RuntimeAssembly { get; }

/// The fully qualified names of the set of referenced assemblies

public string ReferencedAssemblies { get; }

/// The fully qualified names of a folder for temporary files

public string TemporaryFolder { get; }

}

## IProvidedNamespace

### IProvidedNamespace.NamespaceName

string NamespaceName { get }

The namespace string is in the form A.B.C.

The type provider can opt out of top-level namespace resolution by returning null returned from NamespaceName designates global namespace. Empty string is not an allowed return value.

### IProvidedNamespace.ResolveTypeName

Signature:

Type ResolveTypeName(string name);

During name resolution, if the compiler sees and identifier in the form of A.B.C.TypeName that is unresolved by the normal name resolution rules, it will call ResolveTypeName on each provided namespace with name=A.B.C. The name passed to ResolveTypeName will be the simple name TypeName.

ResolveTypeName may return null to indicate that it doesn’t know of the type. Otherwise, it can return an instance of System.Type for the compiler to use.

### IProvidedNamespace.GetNestedNamespaces

Signature:

IProvidedNamespace[] GetNestedNamespaces();

During name resolution, the compiler uses this method to determine the namespace fragments available as suffixes to other namespaces.

Each nested-namespace should return an IProvidedNamespace whose NamespaceName property is an extension of the name of the enclosing namespace, e.g. if the enclosing namespace has name A.B, the nested namespace can have name A.B.C, A.B.D or A.B.C.D. If there are no nested provided namespaces in this namespace, GetNestedNamespaces should return an empty array. null is not a valid return value.

### IProvidedNamespace.GetTypes()

The GetTypes method is used when a namespace is “opened” to determine the names brought into scope, and to provide intellisense completion menus. The GetTypes() method is used for this:

Type[] GetTypes();

If a type in the provided namespace is returned by GetTypes(), it should also be resolved by ResolveTypeName.

ResolveTypeName should not resolve type names that are not reported by GetTypes().

## ITypeProvider

### ITypeProvider.GetNamespaces

Signature:

IProvidedNamespaces[] GetNamespaces();

Return the namespaces provided by this type provider.

### ITypeProvider.GetStaticParameters

Signature:

ParameterInfo[] GetStaticParameters(Type typeBeforeArguments);

During type checking of a type A<*static-arguments*>*opt*, the compiler calls this method to determine if the provided type A accepts static arguments. If so, the arguments are represented by the returned ParameterInfo’s. Each ParameterInfo may be queried for its Name, Kind (ParameterType) and RawDefaultValue. The compiler checks that any static parameters provided in the syntax of the source program match the kinds of those expected.

The static arguments should be returned in position order.

RawDefaultValue may not be null.

### ITypeProvider.ApplyStaticArguments

Signature:

Type AppyStaticArguments(Type typeBeforeArguments,

string typeNameAfterArguments,

obj[] staticArguments);

The type typeBeforeArguments is always one of the types returned from provider.GetTypes or provider.ResolveTypeName.

During type checking of a type A<*static-arguments*>*opt*, the compiler calls this to apply the static arguments to the provided type A. See Section 4.

The method must return a System.Type with name typeNameAfterArguments. This type need never be returned from direct queries to GetTypes or ResolveTypeName – the name is simply a unique identifying name taking into account the static parameters.

### ITypeProvider.GetInvokerExpression()

Eventually, the compiler must reduce the types and method calls that it receives from the type provider into IL for the final assembly. The provider method GetInvokerExpression is called for:

* Uses of methods on provided type definitions (including property get, set, and event operations)
* Uses of constructors on provided type definitions

Expression GetInvokerExpression(MethodBase syntheticMethodBase,

ParameterExpression[] parameters);

GetInvokerExpression is invoked with a MethodBase and a set of dummy parameter expressions indicating place-holders for arguments. The types of these parameters are the unerased types of the parameters.

GetInvokerExpression is used for calls to both erased and non-erased types.

GetInvokerExpression is used for constructor calls on erased types using a replacement System.Expression, just as with method calls on erased types. This means constructor calls on an erased type E don’t have to erase directly to IL constructor calls on erasure. This actually make constructor syntax “new E()” on erased types a little more flexible than constructor syntax on non-erased types (which in F# and C# must always compile to an IL constructor call on precisely the given type)

GetInvokerExpression may return any of these expressions:

* System.Linq.Expressions.MethodCallExpression
* System.Linq.Expressions.NewExpression
* System.Linq.Expressions.ExpressionType.Parameter
  + The parameter must be precisely one of the parameter passed to GetInvokerExpression. If not, a TPV error should be reported.
* System.Linq.Expressions.ExpressionType.Constant
  + The constant must be of type bool, sbyte, byte, int16, uint16, int32, uint32, int64, uint64, IntPtr, UIntPtr, single, double, char, string or decimal. Otherwise, a TPV error should be reported.
* System.Linq.Expressions.ExpressionType.TypeAs
* System.Linq.Expressions.ExpressionType.Conditional
* Otherwise, a TPV error should be reported.

GetInvokerExpression may not return null.

The type of the Expression returned by GetInvokerExpression must be an erased type, and must be equivalent to the erased type of the expression it is replacing.

### ITypeProvider.Invalidate

The type provider is responsible for monitoring the resources that it is projecting into the type system. If those resources change then the type provider should signal this event.

event System.EventHandler Invalidate;

Visual Studio is expected to respond by invalidating and requerying all types that this provider has returned so far.

A typical example of this would be an Excel provider that discovers that there has been a change in the Excel file that is being used.

## Special Attributes

### TypeProviderXmlDocAttribute

The TypeProviderXmlDocAttribute attribute can be added to provided type definitions and provided members. The language service will display the CommentText property from the attribute in the appropriate place when the user hovers over a type or member.

sealed public class TypeProviderXmlDocAttribute : System.Attribute

{

private string commentText;

public string CommentText

{

get { return this.commentText; }

}

public TypeProviderXmlDocAttribute(string comment)

{

this.commentText = comment;

}

}

### TypeProviderDefinitionLocationAttribute

The TypeProviderDefinitionLocationAttribute attribute can be added to provided type definitions and provided members.

sealed public class DefinitionLocationAttribute : System.Attribute

{

public DefinitionLocationAttribute() { }

public string FilePath { get; set; }

public int Line { get; set; }

public int Column { get; set; }

}

A host language service will look for this attribute on types and members. When present, the host will open the given FilePath and jump to the given Line and Column.

### TypeProviderEditorHideMethodsAttribute

Data projections frequently include directory or property-bag objects that are not designed for use as in-memory CLR objects, but are only for projecting further data from. For these objects, the Equals, GetHashCode and other methods provided by the System.Object type can be highly confusing to data-oriented users because these are designed for a programming experience focused on in-memory objects rather than external data and services.

The TypeProviderEditorHideMethodsAttribute attribute can be added to provided type definitions This that a code editor should hide all System.Object methods from the intellisense menus for instances of a provided type.

sealed public class TypeProviderEditorHideMethodsAttribute: System.Attribute

{

public TypeProviderEditorHideMethodsAttribute () { }

}

The language service will look for this attribute on provided types. When present, the language service will not show the Equals, GetHashCode and other System.Object-inherited methods for the type.

### TypeProviderAttribute

The presence of type providers in an assembly is advertised by adding an assembly-level attribute. For example,

[assembly: System.Runtime.CompilerServices.TypeProviderAssembly]

The compiler will check each referenced assembly for this attribute. If the attribute is present, then the compiler will look for types which implement ITypeProvider and which have the type-level attribute:

[System.Runtime.CompilerServices.TypeProvider]

### TypeProviderAssemblyAttribute

The ITypeProvider implementation must be hosted in the runtime of Visual Studio. However, this may not be the runtime of the project being built. For example, the project may target Silverlight, phone or older .NET version.

To support this, Type Providers allow the runtime assembly of the type provider to be separated from the design time assembly. This is done by placing an attribute in the runtime assembly which declares the name of the separate design time assembly.

[<assembly: TypeProviderAssembly("Designer.Assembly")>]

The compiler will search for the design time assembly and use it to load type providers from.

In this case, the type provider must only return System.Type objects corresponding to assemblies relevant to the target platform, using ReflectionOnlyLoadContext if needed.

The argument to TypeProviderAssembly attribute can be an assembly file name, a simple name of a full assembly name, with version number and public key token.

1. If the argument to TypeProviderAssembly attribute ends with “.dll” then it is assumed to be a file name. The compiler will load the assembly from the same directory as runtime assembly that has the TypeProviderAssembly attribute.
2. If the argument TypeProviderAssembly attribute is a full assembly name, the compiler will to load designed assembly from GAC
3. If the argument to TypeProviderAssembly attribute is a simple name, the compiler will attempt to load the assembly with that file name (appending “.dll” extension) from the same directory as runtime assembly, and, failing that, from the GAC.

If no argument or null argument is given then the runtime assembly is used as the design-time assembly.

[<assembly: TypeProviderAssembly>]

### GenerateAttribute (Microsoft.FSharp.Core)

If a provider is generative then:

* If it has a type definition under a path D.E.F, and the .Assembly of that type is in a different assembly A to the provider’s assembly, then that type definition is “generative” or “concrete” type definition.
* In this case, the type definition must first be referenced as follows:

[<Generate>]

type Bing = Samples.WebDirectory.ProgrammableWeb.Bing

See “Generative v. Fully-erasing Providers”

# Type Provider Assemblies and Instances

## Type Provider Design-time Assemblies

A **type provider design-time assembly** is the component loaded into host design-time tools like Visual Studio, F# Interactive and the F# Compiler.

Version: A type provider design-time assembly must target the appropriate CLI version for the host tool. For the F# compiler and Visual Studio, this is .NET 4.0 or later.

Platform: If a type provider design-time assembly has a platform dependency, it must be x86. It is highly preferential that they have no platform dependency.

A type provider design-time assembly should only have the TypeProviderAssemblyAttribute if they are also the runtime assembly for the type provider.

Host tools may

* Use reflection over type provider design-time assemblies, to find .NET type definitions annotated with the TypeProviderAttribute. These are called **type provider design-time instances**.
* Instances may be created as needed by the F# compiler and/or other host process.

Type provider design-time assemblies are loaded into host tools using Assembly.LoadFrom. They are not loaded in a separate AppDomain.

Required characteristics of the assembly are:

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| assembly.GetExportedTypes() |  | Used to get the list of types to look for type provider implementations. |
| typ.GetCustomAttributes |  | Used to search for TypeProviderAttribute on one of the types returned by GetExportedTypes. |
| new ProviderType(configopt) |  | Used to create TypeProvider design-time instances |
| *All others* | n/a | unused |

## Type Provider Runtime Assemblies

A **type provider runtime assembly** is the component referenced in an F# project or script to indicate the use of a type provider.

Version: A type provider runtime assembly can target any version of .NET, Silverlight, Mono, Compact Framework etc. and can be any platform, according to the kind of code being built.

These have an assembly attribute TypeProviderAssemblyAttribute with an argument giving the name of the corresponding design-time assembly. This can be the runtime assembly itself.

This assembly is not loaded by the host, unless it is also a design-time assembly. If the assemblies are different, then it is common for the type provider design-time assembly to use ReflectionOnlyLoadFrom on the runtime assembly and its dependencies.

The host does not use System.Reflection over this assembly.

## Hosting Type Providers: Creation, Lifetime and Disposal

Type Providers are IDisposable and the host calls dispose.

**From the provider-writer’s perspective**:

* Providers can be disposed at any time by the provider host. More than one provider instance can be active in-process with the same configuration parameters (e.g. for multiple scripts and projects using the same provider).
* There is no guarantee that a provider will be disposed, ever. The process may hang on to it for an arbitrarily long time, or rude-exit without disposing it, or just normal-exit without disposing it.
* A provider will only ever be disposed at most once.
* Exceptions during disposal may be silently ignored by the host.

**From our VS language service perspective:**

* Providers are disposed when the containing non-frameworks TcImports is disposed.
* This normally means when the owning IncrementalBuilder is disposed, which means when a IncrementalBuilder is pushed out of our buildCache in service.fs
* Alternatively, when the provider is replaced by another provider in the IncrementalBuilder because some condition in the incremental builder causes a rescan of non-framework assemblies.
* Providers are not necessarily disposed on script or project close – the way the F# background build works means that buildCache entries may hang around longer than that.
* We discard all errors from provider disposal. Disposal can happen at any time, e.g. when  a new project or script starts to build and the project that held the provider gets bumped out of the background build cache. So it’s normal that we don’t have any good place to report the error.

**From the fsc.exe compiler perspective:**

* Providers are not disposed in fsc.exe. Since we don’t get disposal-on-exit in Visual Studio, it doesn’t seem to make sense to do disposal-on-exit for fsc.exe.

## Hosting Type Providers: Threading

All providers involved in the compilation or typecheck-analysis of a single F# assembly are accessed in a single-threaded way from a single compilation thread. This thread does not have a synchronization context (it is “free-threaded”)

# Provided Elements

## Provided Generated Assemblies

These are generated assemblies in the assembly returned by the .Assembly of a generated provided type definition.

Note that an erased provided type definition should return the type provider assembly itself as the .Assembly of the provided type.

### Required Characteristics

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| assembly.GetName() |  |  |
| assembly.FullName |  |  |
| assembly.ManifestModule.FullyQualifiedName | A file path | Used to determine the full name of the assembly file to statically link for a generated assembly. |
| *All others* | n/a | unused |

## Provided Type Definitions

Provided type definitions are those returned by IProvidedNamespace.ResolveTypeName, IProvidedNamespace.GetTypes and recursively by System.Type.GetNestedTypes.

### Required Characteristics of Provided System.Type Definitions

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| typ.Assembly |  | For erased types this should be the type provider assembly itself. |
| typ.BaseType |  |  |
| typ.DeclaringType | null for non-nested  non-null for nested |  |
| typ.FullName |  | Only called for diagnostics |
| typ.Name |  | This may not return a name containing characters that are not allowed in F# and .NET type names, i.e. DOT, PLUS, DOLLAR, AMPERSAND, [, ], /, \, \*, DOUBLE-QUOTE, BACKTICK |
| typ.Namespace |  |  |
| typ.GetAttributeFlagsImpl() |  | Used to by the core System.Type implementation to implement typ.Attributes. [[1]](#footnote-1)   * Erased type definitions must return a value where the special, F#-specific IsErased flag set, value 0x40000000 and given by the F# literal TypeProviderTypeAttributes.IsErased. |
| typ.GetConstructors(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetCustomAttributesData() |  |  |
| typ.GetEvent(string, bindingFlags) |  | Must be consistent with GetEvents, unchecked. bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetEvents(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetField(string, bindingFlags) |  | Must be consistent with GetFields, unchecked. bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetFields(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetInterfaces() | A superset of typ.BaseType.GetInterfaces() |  |
| typ.GetNestedTypes(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly  This method is also called with bindingFlags NonPublic| Public|Instance|Static|DeclaredOnly for all provided generated [<Generated>] types and their nested types. |
| typ.GetMethods(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetProperties(bindingFlags) |  | bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ.GetProperty(name,bindingFlags) |  | Must be consistent with GetProperties, unchecked  bindingFlags are Public|Instance|Static|DeclaredOnly |
| typ. GetEnumUnderlyingType() |  | Only called is .IsEnum returns true |
| typ.IsAbstract |  |  |
| typ.IsClass |  |  |
| typ.IsInterface |  |  |
| typ.IsEnum |  | Erased 🡺 false |
| typ.IsSealed |  | Erased 🡺 false |
| typ.IsValueType |  | Erased 🡺 false |
| typ.ContainsGenericParameters | False |  |
| typ.GetGenericArguments() | [| |] |  |
| typ.HasElementType | False |  |
| typ.IsGenericParameter | False |  |
| typ.IsGenericType | False |  |
| typ.IsGenericTypeDefinition | False |  |
| typ.IsArray | False | Implemented by overriding IsArrayImpl |
| typ.IsByRef | False | Implemented by overriding IsByRefImpl |
| typ.IsPointer | False | Implemented by overriding IsPointerImpl |
| *All others* | n/a | unused |

## Provided Static Parameters

These are ParameterInfo objects returned by GetStaticParameters for a provided type definition.

### Required Characteristics

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| pi.Name |  |  |
| pi.ParameterType |  | must report one of byte, int16, int32, int64, sbyte, uint16, uint32, uint64, decimal, single, double, char, string, bool. |
| pi.RawDefaultValue |  | If a static parameter is optional, this is used to determine the default value for the parameter. |
| pi.Name |  |  |
| pi.Attributes |  | Used to determine if a static parameter is optional |
| pi.GetCustomAttributesData() |  |  |
| *All others* | n/a | unused |

## Provided Methods

These are MethodInfos returned by GetMethod, GetMethods, GetGetterMethod and GetSetterMethod.

### Required Characteristics

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| mi.DeclaringType |  |  |
| mi.GetCustomAttributesData() |  |  |
| mi.GetParameters() |  |  |
| mi.IsAbstract |  |  |
| mi.IsConstructor |  |  |
| mi.IsFinal |  |  |
| mi.IsHideBySig |  |  |
| mi.IsStatic |  |  |
| mi.IsVirtual |  |  |
| mi.Name |  |  |
| mi.ReturnParameter |  |  |
| mi.GetHashCode() |  | Only used for presentational purposes, when filtering VS intellisense lists |
| mi1.Equals(mi2) |  | Only used for presentational purposes, when filtering VS intellisense lists |
| mi.IsGenericMethod | False |  |
| *All others* | n/a | unused |

## Provided Properties

These are PropertyInfos returned by GetProperty and GetProperties.

### Required Characteristics

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| pi.CanRead |  |  |
| pi.CanWrite |  |  |
| pi.DeclaringType |  |  |
| pi.GetCustomAttributesData() |  | required |
| pi.GetGetMethod(nonPublicopt=false) |  |  |
| pi.GetSetMethod(nonPublicopt=false) |  |  |
| pi.GetIndexParameters() |  |  |
| pi.IsInitOnly |  |  |
| pi.IsStatic |  |  |
| pi.Name |  |  |
| pi.PropertyType |  |  |
| pi.GetHashCode(..) |  | Only used for presentational purposes, when filtering VS intellisense lists |
| pi1.Equals(pi2) |  | Only used for presentational purposes, when filtering VS intellisense lists |
| *All others* | n/a | unused |

## Provided FieldInfos

These are FieldInfo objects returned by GetField and GetFields.

### Required Characteristics

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| fi.DeclaringType |  |  |
| fi.FieldType |  |  |
| fi.GetRawConstantValue() |  | fi.IsLiteral |
| fi.GetCustomAttributesData() |  | required |
| fi.IsInitOnly |  |  |
| fi.IsStatic |  |  |
| fi.IsSpecialName | true for enum “value” field, otherwise false |  |
| fi.Name |  |  |
| fi.GetHashCode() |  | Only used for presentational purposes, when filtering VS intellisense lists |
| fi1.Equals(fi2) |  | Only used for presentational purposes, when filtering VS intellisense lists |
| *All others* | n/a | unused |

## Provided EventInfos

These are EventInfo objects returned by GetEvent and GetEvents.

### Required Characteristics of Design-time Provided System.EventInfo’s

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| ev.DeclaringType |  |  |
| ev.EventHandlerType |  |  |
| ev.GetAddMethod() |  |  |
| ev.GetCustomAttributesData() |  |  |
| ev.GetRemoveMethod |  |  |
| ev.Name |  |  |
| ev.GetHashCode() |  | Only used for presentational purposes, when filtering VS intellisense lists |
| ev1.Equals(ev2) |  | Only used for presentational purposes, when filtering VS intellisense lists |
| *All others* | n/a | unused |

## Provided Parameters

### Required Characteristics of Provided ParameterInfos

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| pi.Name |  |  |
| pi.ParameterType |  |  |
| pi.Attributes |  | Used to determine if a parameter is optional |
| pi.IsOut |  |  |
| pi.RawDefaultValue |  |  |
| pi.GetCustomAttributesData() |  |  |
| *All others* | n/a | unused |

## Provided Types

Provided types are those returned by typ.GetGenericArguments for a provided type, prop.PropertyType for a provided property, param.ParameterType for a provided parameter, and meth.ReturnType for a provided method.[[2]](#footnote-2)

### Required Characteristics

These are associated with a root provided type definition and, if a generated type, an associated generated assembly A.

These can be:

* References to types or nested types in other assemblies, such as System.Int32 or System.DateTime in mscorlib.dll. The reference is determined by typ.Namespace, typ.Name, typ.DeclaringType and typ.Assembly.AssemblyName.
* References to erased provided types provided by the same or another type provider.
* References to generated provided types associated with the same assembly A as the provided type.

Below, (\*) indicates the method is not called if IsGenericType, IsArray, IsByref or IsPointer return true. In these cases, either GetElementType() or GetGenericTypeDefinition/GetGenericArguments are called instead to decompose the symbolic type.

|  |  |  |
| --- | --- | --- |
| Member | Return Value | Comments |
| typ.Assembly (\*) |  | For erased types this should be the type provider assembly itself. |
| typ.BaseType (\*) |  |  |
| typ.DeclaringType (\*) | null for non-nested  non-null for nested |  |
| typ.FindInterfaces (\*) |  |  |
| typ.FullName (\*) |  |  |
| typ.Name (\*) |  |  |
| typ.Namespace (\*) |  |  |
| typ.GetGenericTypeDefinition() |  |  |
| typ.GetGenericArguments() |  |  |
| typ.GetElementType() |  |  |
| typ.IsArray |  |  |
| typ.IsGenericType |  |  |
| typ.IsArray |  |  |
| typ.IsByRef |  |  |
| typ.IsPointer |  |  |
| *All others* | n/a | unused |

## Provided Measures

A type provider may provide types with unit-of-measure annotations. This means a provided type may refer to an F# type that accepts unit of measure annotations and give values for those measures. For example, a type provider can provide the following types:

* float<kg>
* MyVector<float,kg / m ^2>

In these cases kg and kg/m^2 are called *provided measures*. For the purposes of the type provider machinery provided measure annotations are themselves represented using System.Type objects.

A **provided type** **with measure annotations** must be a type object with the following characteristics:

* IsGenericType must return true
* The type object returned by GetGenericTypeDefinition() must be one of the following:
  + A type object for one of the primitive types double, single, sbyte, int16, int32, int64 or decimal.
  + A type object for an F# type which accepts unit of measure arguments.

A **provided measure** must be a type object with the following characteristics:

* A measure product, e.g. m \* m
  + GetGenericTypeDefinition() returns typedefof<MeasureProduct<\_,\_>> from Microsoft.FSharp.Core.CompilerServices.
  + GetGenericArguments() returns two type object t1 and t2 where these are provided measures.
* A measure inverse, e.g. / m
  + GetGenericTypeDefinition() returns typedefof<MeasureInverse<\_>> from Microsoft.FSharp.Core.CompilerServices.
  + GetGenericArguments() returns one type object which is itself a provided measures.
  + One Type object t1 from GetGenericArguments() where if itself a provided measure type.
* A measure unit, 1
  + This is precisely typeof<MeasureOne> from Microsoft.FSharp.Core.CompilerServices.
* A type object for an F# measure defined using [<Measure>] in an F# assembly.

### Examples Implementing System.Type Manually

Note: In practice, measures are usually provided by using a higher-level helper API such as TypeProviderEmit that contains appropriate System.Type implementations.

For example, to provide the type float<kg> as the return type of a property, you must provide a type object with the following characteristics:

/// Get the System.Type representing the measure “kg”

let fsharpCore = typedefof<list<int>>.Assembly

let kgType = fsharpCore.Assembly.GetType("Microsoft.FSharp.Data.UnitSystems.SIModule+Kilogram")

type FloatKg() =

inherit Type()

override this.IsGenericType = true

override this.GetGenericArguments() = [| kgType |]

override this.GetGenericTypeDefinition() = typeof<float>

override this.IsArrayImpl() = false

override this.IsByRefImpl() =false

override this.IsPointerImpl() = false

override this.IsPrimitiveImpl() = false

override this.IsCOMObjectImpl() = false

override this.HasElementTypeImpl() = false

/// The implementation of these methods is useful for debugging

override this.FullName = typeof<float>.Name + "[kg]"

override this.DeclaringType = typeof<float>

override this.Name = typeof<float>.Name + "[kg]"

override this.BaseType = typeof<System.ValueType>

override this.ToString() = this.FullName

/// The implementation of other System.Type methods are unused

To provide the type float<kg/m^2>, you must provide a type object with the following characteristics:

open Microsoft.FSharp.Core.CompilerServices

let fsharpCore = typedefof<list<int>>.Assembly

let kgType = fsharpCore.GetType("Microsoft.FSharp.Data.UnitSystems.SIModule+Kilogram")

let meterType = fsharpCore.GetType("Microsoft.FSharp.Data.UnitSystems.SIModule+Meter")

type FloatKgPerMeterSquared() =

inherit Type()

override this.IsGenericType = true

override this.GetGenericArguments() =

let meterSquared =

typedefof<MeasureProduct<\_,\_>>.MakeGenericType [| meterType; meterType |]

let perMeterSquared =

typedefof<MeasureInverse<\_>>.MakeGenericType [| meterSquared |]

let kgPerMeterSquared =

typedefof<MeasureProduct<\_,\_>>.MakeGenericType [| kgType; perMeterSquared |]

[| kgPerMeterSquared |]

override this.GetGenericTypeDefinition() = typeof<float>

To provide an instantiation MyVector<int[],kg> of a user-defined type with a measure parameter, e.g.

type MyVector<'Storage, [<Measure>] 'U>() =

member x.Dummy = 1

you must provide a type object with the following characteristics:

open Microsoft.FSharp.Core.CompilerServices

let fsharpCore = typedefof<list<int>>.Assembly

let kgType = fsharpCore.Assembly.GetType("Microsoft.FSharp.Data.UnitSystems.SIModule+Kilogram")

type FloatKgPerMeterSquared() =

inherit Type()

override this.IsGenericType = true

override this.GetGenericArguments() = [| typeof<int>; kgType |]

override this.GetGenericTypeDefinition() = typedefof<MyVector<\_,\_>>

# Generated Code

## Static Linking of Generated Assemblies

If a type or method is not erased, then the compiler will locate its assembly and attempt to inject the IL from that assembly into its current build output. If that assembly doesn’t have real IL (a file on disk) then the compiler will emit an error.

Because of the use of static linking, the following restrictions apply:

* If the assembly A refers to any assembly B that is not in the reference set of the compilation, a warning is given.
* No two statically linked assemblies within one assembly may define the same named type.

## Type Relocation of Generated Types

Types in generated assemblies are “relocated” during static linking. This means that, given a declaration

namespace Foo.Bar

[<Generate>]

type X = *TypeGenerator*

and (one or more) corresponding generated assemblies A in the generated provided type definitions implied by *TypeGenerator*, the metadata and code of the generated assemblies are rewritten so that all definitions and references of a type T1…TN in A become definitions or references to a type X.T1…TN. This may be a nested type.

In the M3 release of F# 3.0, type relocation may be suppressed by specifying the SuppressRelocation flag in the TypeAttributes of a generated type.

## Dependencies of Generated Code

Generated code consists of

* The .NET IL emit of the expressions returned by GetInvokerExpression (note, after any F# optimization)
* The static linking of any generated assemblies.

As a result, generated code only depends on these elements. If, for example, GetInvokerExpression only ever returns constant values for all erased provided types, then the IL code generated will not have any additional dependencies.

In particular, it is normal that generated code has no dependencies on the design-time type provider DLL. Even when a type provider uses the one DLL for both design-time and run-time, it is normal that a type provider only emits calls to a simple Runtime API portion of the type provider DLL.

## Support Embedding Resources

Embedded resources in a generated assembly are also included in the final assembly.

Linked resources can also be included in a generated assembly, but are not recommended for use because the build, debug and deployment machinery for Visual Studio will not include deploying the linked resources (the files linked to by the resources are not included as build outputs of the Visual Studio project).

Note: This could be used, to, for example, include a goto\action table for a parser, or error message text for a strongly typed error provider, or .resource files.

# Exceptions and Diagnostics from Type Providers

All uses of all members from provided types may throw exceptions. In all cases, if a type provider throws an exception, a user error must be reported attributing the error to a specific type provider.

Type provider exceptions never result in internal compiler errors (i.e. Watson dumps).

Type providers may not report warnings.

Type providers may throw the following exceptions to deliberately report errors. In all cases the Message field will be used as the error text, and no stack trace will be shown to the user.

* System.NotSupportedException
* System.IO.IOException
* System.Exception

# Cross Assembly References

## Cross Assembly References for Erased Types

A compiled F# assembly may include a reference to an erased type at the assembly boundary.

## Cross Assembly References for Generated Types

A compiled F# assembly may include a reference to a generated type at the assembly boundary.

# Intellisense, Quick Info, Method Tips

## QuickInfo

See separate speclets by Brian McNamara on Intellisense, ParameterInfo and QuickInfo for type providers.

# Deploying Type Providers

## Design-time Loader Resolution for Type Provider DLLs and Dependencies

Type provider design time DLLs are loaded by the following rules:

1. If the assembly name ends with .dll, we look in the directory next to runtime assembly.

This uses an Assembly.LoadFrom call from the resolution context used by fsc.exe, fsi.exe or devenv.exe.

1. If the assembly name is more than a simple name, i.e. version number or public key token or both, then a GAC resolution is used.

This is done by using Assembly.Load from the resolution context used by fsc.exe, fsi.exe or devenv.exe.

1. If the assembly name is a simple name, we look in the directory next to run-time assembly and then in the GAC

TypeProvider design-time DLLs are only ever x86 32-bit or AnyCPU. This is because fsc.exe and fsi.exe for .NET 4.0 is x86. NOTE: this may cause problems in scenarios where fsi.exe runs as x64. Note: should set the x86 flag in any templates we ship?

## xcopy installation

Type providers can be installed and referenced using xcopy installation. A typical layout will be with all provider DLLs in one directory:

Runtime DLLs:

DataStore.TypeProvider.dll

DataStore.TypeProvider.WindowsPhone.Silverlight.v4.0.dll

DataStore.TypeProvider.Silverlight.v3.0.dll

DataStore.TypeProvider.Silverlight.v4.0.dll

DataStore.TypeProvider.Xbox360.CompactFramework.v3.7.dll

Design-time DLLs:

DataStore.TypeProvider.DesignTime.dll

There is no obligation to use a single design-time DLL for all platforms.

Type providers can be written to be cross-platform (Mono/.NET) and can be used from Mono, unchanged.

Neither runtime nor design-time type provider DLLs need to be installed in the GAC.

Neither runtime nor design-time type provider DLLs need a strong name.

Note: For scenarios where F# compiler/language-service is hosted in Silverlight, the type provider design-time DLL will need to be recompiled to target Silverlight. This is not a scenario for dev11, though may be an OOB scenario we support through the OSS release and through Microsoft External Research.

## Installation into AssemblyFolders

Extension assemblies can be installed anywhere on a machine, but if you want them found by the “Add Reference” dialog and auto-resolved from simple names by F# scripts and msbuild, then the type provider installation process should add a registry key under:

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\.NETFramework\v4.0.30319\AssemblyFoldersEx

This should refer to a directory holding the appropriate .NET 4.0 runtime DLL. If a separate design-time DLL is used then the design-time DLL should be placed alongside this.

For Silverlight, the registry key should be under

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Silverlight\v4.0\AssemblyFoldersEx

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Silverlight\v5.0\AssemblyFoldersEx

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Microsoft SDKs\Silverlight\v6.0\AssemblyFoldersEx

Etc.

## Referencing Type Providers

A user project should only reference the appropriate runtime DLL for a type provider.

For non-.NET-4.0+ platforms (e.g. Windows Phone 7 XNA, or Silverlight), a runtime DLL is always required to act as a referenced assembly at design time.

Note: this is the case even if the “runtime” DLL is not actually needed at runtime, i.e. if no GetInvokerExpression calls or erased type representations ever refer to the runtime DLL. In this case this “runtime” DLL would simply include a single attribute giving the name of the appropriate design-time DLL.

# Security and Threat Modeling

See separate document.

# Debugging and Profiling Type Providers

* devenv /debugexe fsc.exe -r:TypeProvider.dll someInput.fsx
* Turn on “catch first chance exceptions”

As part of this, we should endeavor to remove all first chance exception raising on the startup/normal-execution of the F# compiler. This is not easy, particularly for the ReferenceResolution code, and in lieu of this we should record this as a known issue in UE documentation on debugging type providers.

# Cut Features

## Type Provider Isolation & Unloading

Based on experience of authoring and debugging providers, putting providers in a separate, unloadable app domain is very, very tempting. However we think we can ship the feature without doing that.

## Managing Build Dependencies and Files

Type providers may reference files that aren’t directly visible to the build system. If those files change then a rebuild is needed the next go-round. C++ has a similar problem because it may end up including files based on preprocessor results. One proposed solution was that we spawn their compile under FileWatch.dll which gathers inputs and outputs. And do the same in the fsc build task. Contacts: Sara Joiner, Kieran Mockford

*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.*

I would prefer we don’t architect in anything extra than what we already have. Or at least nothing more than something very simple like a “please use this directory for intermediate files” in the provider context.

**Resolution (cut)**: Add an intermediate-files directory to the type provider context

Q: 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?

Except where the proposed additions to MSBuild support are involved, I’d say this is largely in the realm of provider design, hence not our M2 focus. Just don’t worry about it. We also have time for this in M3.

**Resolution (cut)**: implement MSBuild file-touch tracking, as planned.

## Watson Dumps sent to Type Provider Authors

We considered doing the engineering to allow type provider exceptions to be routed a Watson dumps to type provider authors. We won’t do this for F# 3.0.

## Supported SDK for writing type providers

The current position is that we do not plan to ship any framework for writing providers, apart from samples. A simple, clear architecture, well documented interfaces and a good sample set is what we need, and we have all the way through to Beta1, Beta2 and RTM to complete those.

## Binary Compatibility and Type Providers

Q: What happens when a TP changes implementation and erases more types?

That would be a binary & source breaking change. Overall I don’t see us venturing into concrete types inheriting from erased types. Indeed, the original position was that erased types were always sealed, and it’s worth remembering that that would still be a valid position for most of our goals. However inheritance (including multiple inheritance) is common enough in data models that we should really allow them.

## ResolveExtensionReference

Assembly ResolveExtensionReference(string name);

What different places can the “name” come from?

fsc.exe -r “MyDatabase.dbml”

#r “some-connection-string”

<ExtensionReference> d:\SomeDir\Invoice.csv</ExtensionReference> in the .fsproj

Q. Is the returned Assembly just a standard assembly – or – type provider assembly?

A: Type provider assembly

Q: How does the TP assembly know about the (name:string)that triggered it?

A: It is passed in to ResolveExtensionReference, the resulting System.Assembly returned by ResolveExtensionReference can remember it in whatever proprietary way it likes.

## Architecting a Live Connection to Visual Studio Buffers

*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?*

We do not do anything here. A provider can work it out via a VS API if it wishes.

**Resolution**: No action

## F#-isms

Provided type definitions have no way to report F#-isms

* definitions of records
* definitions of unions
* definitions of active patterns
* F#-specific generic constraints
* Modules (which can be opened)
* Non-nullability
* The special attributes the compiler takes regard of
* Extension member attributes
* CompilerMessageAttribute (an F# specific attribute)
* ExperimentalAttribute (an F# specific attribute)
* UnverifiableAttribute (an F# specific one)
* AutoOpenAttribute (an F# specific one)

# Risk Mitigation

Features will be cut in the following order:

* Abstract members in erased types
* Erased interface types
* Unsealed erased types

1. In M3, generated type definitions may also return a value indicating that type relocation is suppressed, value 0x80000000, the F# literal TypeProviderTypeAttributes.SuppressRelocation. [↑](#footnote-ref-1)
2. Note: do not confuse provided types and provided type definitions. See above for provided type definitions. Provided types are just the algebra of types appearing in the signatures of the elements of provided type definitions, and don’t necessarily include references to provided type definitions. [↑](#footnote-ref-2)