Writing F# Type Providers

Contents

[Introduction 2](#_Toc304468853)

[1 Before You Begin 2](#_Toc304468854)

[2 A Simple Erasing Type Provider 3](#_Toc304468855)

[2.1 Using the Provider 4](#_Toc304468856)

[2.2 Recompiling the Provider 4](#_Toc304468857)

[2.3 Debugging the Provider 5](#_Toc304468858)

[2.4 The Type Provider Implementation 5](#_Toc304468859)

[2.5 Providing One Type and its Members 6](#_Toc304468860)

[2.6 More on Erased Provided Types 8](#_Toc304468861)

[2.7 Choosing Representations for Erased Provided Types 9](#_Toc304468862)

[3 Example: A Type Provider Using Data from Static Parameters 10](#_Toc304468863)

[3.1 Type Checked Regex Provider 10](#_Toc304468864)

[4 Example: A Type Provider Backed By Local Data 13](#_Toc304468865)

[4.1 Mini CSV File Provider 13](#_Toc304468866)

[5 Going Further 15](#_Toc304468867)

[5.1 Design and Naming Conventions for Type Providers 15](#_Toc304468868)

[5.1.1 Providers for Connectivity Protocols 15](#_Toc304468869)

[5.1.2 Utility Providers for General Coding 16](#_Toc304468870)

[5.1.3 Singleton Data Sources 16](#_Toc304468871)

[5.2 The GetConnection Design Convention 16](#_Toc304468872)

[5.3 Creating Types and Members On-demand 16](#_Toc304468873)

[5.4 Adding Static Parameters 16](#_Toc304468874)

[5.5 Providing Array, ByRef and Pointer types 17](#_Toc304468875)

[5.6 Providing Unit of Measure Annotations 17](#_Toc304468876)

[5.7 Accessing Project-Local or Script-Local Resources 17](#_Toc304468877)

[5.8 Invalidation 17](#_Toc304468878)

[5.9 Caching Schema Information 17](#_Toc304468879)

[5.10 Type Providers Backed By Remote Data and Services 17](#_Toc304468880)

[5.11 Exceptions and Diagnostics from Type Providers 18](#_Toc304468881)

[5.12 Some Rules For Type Providers 18](#_Toc304468882)

[5.12.1 Provided types must be reachable 18](#_Toc304468883)

[5.13 Development Tips 18](#_Toc304468884)

[5.13.1 Run Two Visual Studio Instances 18](#_Toc304468885)

[5.13.2 Debug type providers by using invocations of fsc.exe 18](#_Toc304468886)

[5.14 Some limitations of the F# Type Provider Mechanism 19](#_Toc304468887)

[5.15 Some limitations of the ProvidedTypes support code 19](#_Toc304468888)

# Introduction

This doc covers the basics of how to write F# type providers and some of the design issues associated with providers.

For the Developer Preview Release, the ProvidedTypes helper code used on this article can be found as part of the F# 3.0 Sample Pack at <http://fsharp3sample.codeplex.com/>

# Before You Begin

The F# Type Provider mechanism is a powerful addition to the F# language and tooling. We expect a fair degree of experimentation with this mechanism, which is somewhat inevitable.

The F# Type Provider mechanism is primarily designed for injecting stable data and service information spaces into the F# programming experience.

* It is not designed for injecting information spaces whose schema changes during program execution in ways that are relevant to program logic.
* It is not designed for intra-language meta-programming (though there are some valid uses in that domain)

The mechanism is one that should only be used “where necessary” and where there is very high value accrued from the development of a type provider.

We encourage users to be particularly vigilant for people writing a type provider where a schema is not available. For example, someone might say “I want a type provider for JSON” – that’s fine, but you’re going to need some schema information, probably inferred from a sample JSON blob. That might be ok, but equally it might be a bit clumsy.

Likewise, we encourage users to be vigilant for people writing a type provider where an ordinary (or even an existing) .NET library would suffice.

Some basic questions to ask before you begin are

* Do you have a schema? If so, what’s the mapping into the F# and .NET type system?
* Do you have “enough” instances of the data/information space to make it worth your while to write a type provider? Would a normal .NET library suffice?
* How fast will your schema change?
  + Will it change during coding?
  + Will it change between coding sessions?
  + Will it change during program execution?

Type providers are best suited to places where the schema is stable at runtime.

# A Simple Erasing Type Provider

This sample is “**Samples.HelloWorldTypeProvider**” in the “SampleProviders\Providers” directory of the F# 3.0 Sample Pack, at <http://fsharp3sample.codeplex.com>.

The provider makes available a “type space” containing 100 erased types, as follows (using F# signature syntax, and omitting the details for all except Type1):

namespace Samples.HelloWorldTypeProvider

type Type1 =

/// This is a static property

static member StaticProperty : string

/// This is a constructor taking no arguments

new : unit -> Type1

/// This is a constructor taking one argument

new : data:string -> Type1

/// This is an instance property

member InstanceProperty : int

/// This is an instance method

member InstanceMethod : x:int -> char

/// This is an instance property

nested type NestedType =

/// This is StaticProperty1 on NestedType

static member StaticProperty1 : string

…

/// This is StaticProperty100 on NestedType

static member StaticProperty100 : string

type Type2 =

…

…

type Type100 =

…

Note the set of types and members provided is statically known – this example doesn’t leverage any of the ability of providers to provide many (or infinite) types, nor types dependent on a schema.

The implementation of the type provider is outlined below. We walk through the details in later sections.

namespace Samples.FSharpPreviewRelease2011.HelloWorldTypeProvider

open System

open System.Reflection

open Samples.FSharpPreviewRelease2011.ProvidedTypes

open Microsoft.FSharp.Core.CompilerServices

open Microsoft.FSharp.Quotations

[<TypeProvider>]

type SampleTypeProvider(config: TypeProviderConfig) as this =

inherit TypeProviderForNamespaces()

let namespaceName = "Samples.HelloWorldTypeProvider"

let thisAssembly = Assembly.GetExecutingAssembly()

let makeOneProvidedType (n:int) =

…

let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]

do()

## Using the Provider

To use this provider, open a **separate** instance of Visual Studio 11 and reference the provider from a script using #r, e.g.

"C:\Program Files (x86)\Microsoft Visual Studio 11\Common7\IDE\devenv.exe" script.fsx

where the script contains:

#r @".\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty

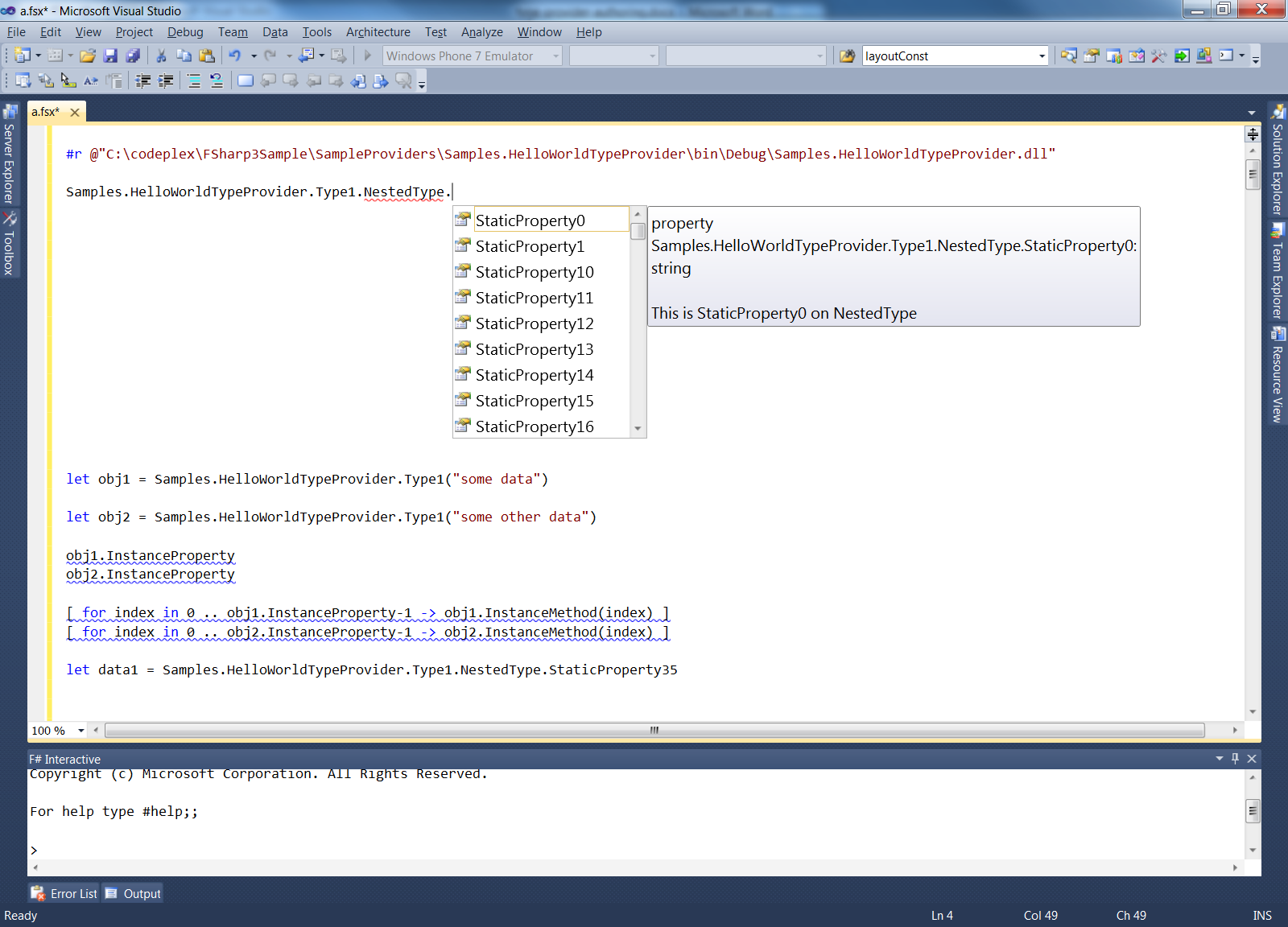
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]

[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35

Then look for the types under Samples.HelloWorldTypeProvider. You will see IntelliSense menus as follows:



## Recompiling the Provider

When recompiling the provider, make sure you have exited all instances of Visual Studio and F# Interactive using the provider DLL before recompiling the provider.

## Debugging the Provider

To debug this provider using 'print' statements, make a script that exposes a problem with the provider, then use

fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx

To debug this provider using Visual Studio, use the following from an administrator-enabled Visual Studio command prompt.

devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx

Be sure to disable "Just My Code" debugging and consider setting first-chance exception catching using

Debug --> Exceptions --> CLR Exceptions --> Thrown

## A Look at the Compiled Code

ILDASM

## The Type Provider Implementation

In this section, we walk through the principle sections of the type provider implementation.

Define the type provider type:

[<TypeProvider>]

type SampleTypeProvider(config: TypeProviderConfig) as this =

Next, inherit a default implementation of the ITypeProvider interface. This helper base type comes from the ProvidedTypes API and is suitable for providing a finite collection of eagerly provided namespaces containing a finite number of fixed, eagerly provided types:

inherit TypeProviderForNamespaces()

Next, define local private values that specify the namespace for the provided types, and fetch the type provider assembly itself (erased types are logically associated with the type provider assembly):

let namespaceName = "Samples.HelloWorldTypeProvider"

let thisAssembly = Assembly.GetExecutingAssembly()

Next, a function to provided one of the types Type1…Type100. We look at this function in more detail below.

let makeOneProvidedType (n:int) = …

Next, generate the 100 provided types:

let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

Next, add the types as a provided namespace:

do this.AddNamespace(namespaceName, types)

Finally, add an assembly attribute indicating this is a type provider DLL:

[<assembly:TypeProviderAssembly>]

do()

## Providing One Type and its Members

The makeOneProvidedType function does the real work of providing one of the types.

let makeOneProvidedType (n:int) =

…

Let’s walk through the implementation of this function. First, we create the provided type, e.g. Type1, when n = 1, or Type57, when n = 57.

let t = ProvidedTypeDefinition(thisAssembly,

namespaceName,

typeName = "Type" + string n,

baseType = Some typeof<obj>,

IsErased = true)

There are some important things to note here

* This is an **erased** provided type. This means that in compiled code will appear as type 'obj'.
* When specifying a non-nested type, we must specify the assembly and namespace. For erased types, the assembly should be the type provider assembly itself.

We next add XML documentation to the type. This documentation is “delayed”, i.e. computed on-demand, if needed by the host compiler.

t.AddXmlDocDelayed (fun () -> sprintf "This provided type %s" ("Type" + string n))

We next add a provided static property to the type:

let staticProp = ProvidedProperty(propertyName = "StaticProperty",

propertyType = typeof<string>,

IsStatic=true,

GetterCode= (fun args -> <@@ "Hello!" @@>))

A get of this property will always evaluate to the string "Hello!".The GetterCode for the property returns an F# quotation. This represents the code generated by the host compiler for a get of the property.

We add XML documentation to the property.

staticProp.AddXmlDocDelayed(fun () -> "This is a static property")

We now attach the provided property to the provided type. A provided member must be attached to one and only one type, if it is not it will never be accessible.

t.AddMember staticProp

We now create a provided constructor. The constructor takes no parameters.

let ctor = ProvidedConstructor(parameters = [ ],

InvokeCode= (fun args -> <@@ "The object data" :> obj @@>))

The InvokeCode for the constructor returns an F# quotation. This represents the code generated by the host compiler for a get of the property. For example, a use of this constructor

**new Type10()**

will create an instance of the provided type with underlying data "The object data".

We add XML documentation to the constructor and add the provided constructor to the provided type:

ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

t.AddMember ctor

We now create a provided constructor taking one parameter:

let ctor2 =

ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],

InvokeCode= (fun args -> <@@ (%%(args.[0]) : string) :> obj @@>))

The InvokeCode for the constructor returns an F# quotation. This represents the code generated by the host compiler for a call to the method. For example, a use of this contructor:

**new Type10("ten")**

will create an instance of the provided type with underlying data "ten".

You may have already noticed that InvokeCode is a function returning a quotation. An expression representing the parameter value is available in args.[0]. The code for a call to the constructor coerces this parameter to the erased type 'obj'.

After adding the second provided constructor to the type, we now create a provided instance property:

let instanceProp =

ProvidedProperty(propertyName = "InstanceProperty",

propertyType = typeof<int>,

GetterCode= (fun args ->

<@@ ((%%(args.[0]) : obj) :?> string).Length @@>))

A get of this property will evaluate to the length of the string which is the representation object.

The GetterCode returns an F# quotation giving the code generated by the host compiler for a get of the property. Like InvokeCode, GetterCode is a function returning a quotation – the host compiler calls this function with an expression representing the instance object supplied as args.[0].

The implementation of GetterCode then splices into the result quotation at the erased type 'obj',and a cast used to 'prove' that the object is a string.

The next part of makeOneProvidedType provides an instance method with one parameter.

let instanceMeth =

ProvidedMethod(methodName = "InstanceMethod",

parameters = [ProvidedParameter("x",typeof<int>)],

returnType = typeof<char>,

InvokeCode = (fun args ->

<@@ ((%%(args.[0]) : obj) :?> string).Chars(%%(args.[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")

// Add the instance method to the type.

t.AddMember instanceMeth

Finally, we create a nested type, with 100 nested properties. The creation of this nested type and its properties is “delayed”, i.e. computed on-demand.

t.AddMembersDelayed(fun () ->

let nestedType = ProvidedTypeDefinition(typeName = "NestedType",

baseType = Some typeof<obj>,

IsErased=true)

nestedType.AddMembersDelayed (fun () ->

let staticPropsInNestedType =

[ for i in 1 .. 100 do

let valueOfTheProperty = "I am string " + string i

let p = ProvidedProperty(propertyName = "StaticProperty" + string i,

propertyType = typeof<string>,

IsStatic=true,

GetterCode= (fun args -> <@@ valueOfTheProperty @@>))

p.AddXmlDocDelayed(fun () ->

sprintf "This is StaticProperty%d on NestedType" i)

yield p ]

staticPropsInNestedType)

[nestedType])

## More on Erased Provided Types

The example in this section provides only **erased provided types**.

Erased provided types are particularly useful in the following situations:

* When writing a provider for an information space that “just” contains data and methods, or,
* When writing a provider where accurate runtime-type semantics is not critical for practical use of the information space, or
* When writing a provider for an information space which is so large and inter-connected that it is not technically feasible to generate real .NET types for the information space.

In this example, each provided type is erased to type ‘obj’ and all uses of the type will appear as type ‘obj’ in compiled code. In these examples the underlying objects are in fact strings, but in .NET compiled code, the type will appear as **System.Object**. As with all uses of type erasure, expliciting boxing, unboxing and casting can be used to subvert erased types. In this case, an invalid cast exception may result when the object is used. A provider runtime can define its own private representation type to help protect against false representations.

It is not possible to define erased types in F# itself – only provided types may be erased types.

If you choose to use erased types for your type provider (or are using a provider which provides erased types), then you must understand the ramifications, both practical and semantic. An erased type has no “real” .NET type. This means:

* It is not possible to do accurate reflection over the type, and
* It may be possible to subvert erased types by using backdoor runtime casts and other techniques that rely on exact runtime type semantics. Subversion of erased types will almost certainly result in later runtime type cast exceptions.

## Choosing Representations for Erased Provided Types

For some uses of erased provided types, no representation is required – for example, where the erased provided type only contains static properties and members, and contains no constructors, and no methods or properties return an instance of the type.

If it is possible to reach instances of an erased provided type, then you must have in mind two things

* **What is the erasure of a provided type?**

The erasure of a provided type is how the type appears in compiled .NET code. In the F# 3.0 Developer Preview,

* + The erasure of a provided erased class type is always the first non-erased base type in the inheritance chain of the type.
  + The erasure of a provided erased interface type is always ‘object’.
* **What are the representations of a provided type?**

The set of possible objects for an erased provided type are called the *representations* of the provided type. In the example in this document, the representations of all the erased provided types Type1..Type100 are always string objects.

All representations of a provided type must be compatible with the erasure of the provided type. (If not, either the F# compiler will give an error for a use of the type provider, or invalid, unverifiable .NET code will be generated. A type provider returning code which gives an invalid representation is an invalid type provider.)

There are two very common approaches to choosing a representation for provided objects.

1. If you are simply providing a strongly typed wrapper over an existing .NET type, it often makes sense for your type to erase to that type. This is appropriate when most of the existing methods on that type still make sense when using the strongly typed version.
2. If you want to create an API which differs significantly from any existing .NET API, then it makes sense to create new runtime types which will be the type erasure and representations for the provided types.

The example in this document uses strings as representations of provided objects. Frequently, it may be appropriate to use other objects for representations. For example, you may choose to use a dictionary as a property bag:

ProvidedConstructor(parameters = [],

InvokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))

Alternatively, you may define a type in your type provider which will be used at runtime to form the representation, along with a number of runtime operations:

type DataObject() =

let data = Dictionary<string,obj>()

member x.RuntimeOperation() = data.Count

Provided members will construct an instance of this object type:

ProvidedConstructor(parameters = [],

InvokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))

In this case, provided types may (optionally) choose to use this as the type erasure by specifying this type as the baseType when constructing the ProvidedTypeDefinition:

ProvidedTypeDefinition(…, baseType = Some typeof<DataObject>,… )

…

ProvidedConstructor(…, InvokeCode = (fun args -> <@@ new DataObject() @@>), …)

Key Lessons

In this section you have learned how to create a simple erasing type provider which provides a range of types, properties and methods.

# Example: A Type Provider Using Data from Static Parameters

The ability to parameterize type providers by static data enables many interesting scenarios, even in cases when the provider does not need to access any local or remote data. In this section you’ll learn some basic techniques for putting together such a provider.

## Type Checked Regex Provider

Imagine that you want to implement a regular expression type provider which wraps the .NET Regex libraries in an interface which provides some compile-time guarantees, including:

1. Checking that a regular expression is valid
2. Providing named properties on matches based on any group names in the regular expression.

Using type providers, we can create a RegExProviderType type which is parameterized by the regular expression pattern to provide these benefits. The compiler will give us an error if the supplied pattern is invalid, and the type provider can extract the groups from the pattern so that we’ll be able to access them using named properties on matches.

One good way to design a type provider is to consider how the type provider’s exposed API should look to end users, and how this will translate to .NET code.

Here’s one way we might use such an API to get the components of the area code:

type T = RegExProviderType<"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{7}$)">

let reg = T()

let result = T.IsMatch("425-1232345")

let r = reg.Match("425-1232345").Group\_AreaCode.Value //r equals 425

We can observe the following characteristics:

1. The parameterized RegExProviderType type erases to the standard Regex type.
2. The RegExProviderType constructor erases to a call to the Regex constructor, passing in the static type argument for the pattern.
3. The results of the Match method erase to the Match type.
4. Each named group results in a provided property, and accessing the property erases to a use of an indexer on a Match’s Groups collection.

Here is the core of the logic to implement such a provider (see the sample for the full code):

open System

open System.Linq.Expressions

open System.Reflection

open Microsoft.FSharp.Core.CompilerServices

open Microsoft.FSharp.Collections

open System.Collections.Generic

open System.Text.RegularExpressions

type HelperClass() = <contains some static members>

type DebugType() 1=

inherit System.Object()

[<TypeProvider>]

type public RegExTypeProvider() as this =

inherit TypeProviderForNamespaces()

let thisAssembly = Assembly.GetExecutingAssembly()

let rootNamespace = "<your namespace for the generated type>"

let baseType = typeof<DebugType>

let regexType =

let t = ProvidedTypeDefinition2(thisAssembly, rootNamespace,

"<type provider type name>3", Some baseType)

t.DefineStaticParameters([ProvidedStaticParameter("pattern",typeof<string>)],

fun theTypeNameOfThisInstantiation args ->

match args with

| [| :? string as template |]5 ->

*<add the Match, IsMatch, and the embedded type>*

t

| \_ -> failwith "need generic definition"

)

*<add the method or property for RegExTypeProvider>*

t

do this.AddNamespace(rootNamespace, [regexType] )

[<assembly:TypeProviderAssembly>]

do()

In the example:

1. A DebugType is defined and this can help to identify the problem. Before the code is released, be sure to make it System.Object or other type.
2. Define the type provider type, this type is called “t”
3. The type provider type name. in our case, it is RegExProviderType.
4. Define the input string. Its type is string and name is “pattern”.
5. Take the value of input string as variable “template”. This is where we can access the pattern.

The type defined above is no use without any meaningful methods or property. Here is the way to add a method isMatch:

let mi =

ProvidedMethod(methodName="IsMatch",

parameters=[ ProvidedParameter("data", typeof<string>) ],

returnType=typeof<bool>,

IsStaticMethod=true,

InvokeCode= (fun args -> <@@ (Regex.IsMatch((%%args.[0]:string), template)) @@>))

mi.AddXmlDoc("Test if the input string matches the pattern")

t.AddMember mi

The above code defines a method “IsMatch”. It takes string as input and returns typeof<bool>.

The only tricky part is the %%args notation. If the method is a static method, the args is the parameter list. The first element in the args is the first parameter. If the method is a non-static method, the first argument is “this” pointer. The args is a list of “this” + parameter list. Please make sure the InvokeCode’s return type agrees to the type provided to ProvidedMethod.

We now add a provided property:

let pi = ProvidedProperty("RawRegex",

propertyType=typeof<Regex>,

IsStatic=true,

GetterCode=(fun args -> <@@ new Regex(template) @@>))

pi.AddXmlDoc "Get the raw regular expression"

t.AddMember pi

The code above defined a Regex type property called “RawRegex”. It return the Regex instance initialized from the variable “template”. Because the regular expression type provider gets the pattern input string from outside. The variable called “template” is the input string; this is a critical channel for a type provider to get information from outside world.

At this point we have enough for a simple type provider. The tricky part is how to get Group\_AreaCode from the function call. Because the Group\_AreaCode is a property generated from the input string, this property must be the property from a generated type like RegExTypeProviderType. At this point, your intuition might already tell you that another ProvidedTypeDefinition is needed. Your intuition is correct.

let matchType =

let t = ProvidedTypeDefinition("MatchEx", baseType=Some typeof<Match>)

…

t

The new type is defined as “MatchEx” and based on Match. The reason we define this type is because we want to add group name to this type. The only place we can access input string is inside the “DefineStaticParameters” function. As a result, it makes sense to create the provided type and property in that function.

let reg = new Regex(template)

for name in reg.GetGroupNames() do

if name <> "0" then

let propertyName = sprintf "Group\_%s" name

let pi =

ProvidedProperty(propertyName,

propertyType=typeof<Group>

GetterCode= (fun args -> <@@ ( (%%args.[0]:Match).Groups.[name] ) @@>))

matchType.AddMember pi

 The next step is to change the Match function’s return type to this newly created type “MatchEx”.

let mi =

ProvidedMethod("Match",

parameters=[ ProvidedParameter("data", typeof<string>)],

returnType=matchType,

InvokeCode = (fun args -> <@@ (Regex(template).Match((%%args.[1]:string))) @@>)

mi.AddXmlDoc("Match function")

t.AddMember mi

There are a few key points to note:

* Types with static parameters are created using the DefineStaticParameters method on the ProvidedType class.
* If an invalid regular expression is used as the static parameter (e.g. with mismatched parentheses), then when the type provider tries to create an instance it will throw an exception, which will be shown to the user.
* A provided type can contain constructors, methods, and properties, which are instances of the ProvidedConstructor, ProvidedMethod, and ProvidedProperty types, respectively.
* If a provided property’s GetterCode property is set then the property can be read. If its SetterCode is set then it can be written to.
* We are using a nested Match type within the parameterized RegExTypeProviderType<\_> type so that we can use the pattern that was used as a static argument to generate the group properties. Nested types are again instances of the ProvidedType type.
* We use the AddXmlDoc method so that IntelliSense will provide usage information to users of the provided methods and properties.

Key Lessons

In this section you have learned how to create a type provider which operates on its static parameters. The provider checks the static parameter and provides operations based on its value.

# Example: A Type Provider Backed By Local Data

Frequently we want type providers to present APIs based not only on static parameters, but also information found on local or remote systems. In this section, we’ll consider type providers based on local data, such as local data files.

## Mini CSV File Provider

As a simple example, we’ll consider a type provider for accessing scientific data in CSV format. We’ll assume that we are accessing CSV files that contain a header row followed by floating point data. Given data like this:

|  |  |
| --- | --- |
| Distance (metre) | Time (second) |
| 50.0 | 3.7 |
| 100.0 | 5.2 |
| 150.0 | 6.4 |

We want to provide a type which allows us to get rows with a Distance property of type float<metre> and a Time property of type float<second>.

For simplicity, we will make the following assumptions:

* Header names are either unit-less or have the form “Name (unit)”, and do not contain commas.
* Units are all SI units using the names found in the Microsoft.FSharp.Data.UnitSystems.SI.UnitNames module.
* Units are all simple (e.g. metre) rather than compound (e.g. metre/second)
* All columns contain floating point data.

A more complete provider would loosen these restrictions.

Again, we want to consider how the API should look. Given an info.csv file with the contents from the table above (in comma separated format), we want to be able to write something like this:

let info = new MiniCsv<"info.csv">()

for row in info.Data do

let (time:float<second>) = row.Time

printfn "%f" (float time)

In this case we expect the compiler to convert these calls into something like this:

let info = new MiniCsvFile("info.csv")

for row in info.Data do

let (time:float) = row.[1]

printfn "%f" (float time)

Here are some notes on this translation:

* It will require us to define a real CsvFile type in the type provider’s assembly. It is common for type providers to rely on a few helper types and methods to wrap important logic.
* Since measures are erased at runtime, we can use a float[] as the erased type for a row. Even though the compiler will treat different columns as having different measure types (e.g. in our example the first column has type float<metre> and the second has float<second>), the erased representation can remain quite simple.

Here’s the core of the implementation:

type CsvFile(filename) =

let data =

seq { for line in File.ReadAllLines(filename) |> Seq.skip 1 do

yield line.Split(',') |> Array.map float }

|> Seq.cached

member \_\_.Data = data

[<TypeProvider>]

type public MiniCsvProvider() as this =

inherit TypeProviderForNamespaces()

let asm = System.Reflection.Assembly.GetExecutingAssembly()

let ns = "CsvTypeProvider"

let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

let filename = ProvidedStaticParameter("filename", typeof<string>)

do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

let headerLine = File.ReadLines(filename) |> Seq.head

let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))

let headers = Regex.Matches(headerLine, "(?<header>[^,]+)(,|$)")

for i in 0 .. headers.Count - 1 do

let headerText = headers.[i].Groups.["header"].Value

let fieldName, fieldTy =

let m = Regex.Match(headerText, @"(?<field>.+) \((?<unit>.+)\)")

if m.Success then

let unitName = m.Groups.["unit"].Value

let units = ProvidedMeasureBuilder.Default.SI unitName

(m.Groups.["field"].Value,

ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>, [units]))

else

headerText, typeof<float>

let prop = ProvidedProperty(fieldName, fieldTy,

GetterCode = fun [row] -> <@@ (%%row:float[]).[i] @@>)

prop.AddDefinitionLocation(0, headers.[i].Index, filename)

rowTy.AddMember(prop)

let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

ty.AddMember(ProvidedConstructor([], InvokeCode = fun [] -> <@@ CsvFile(filename) @@>))

ty.AddMember(ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],

InvokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>))

ty.AddMember(ProvidedProperty("Data", typedefof<seq<\_>>.MakeGenericType(rowTy),

GetterCode = fun [csvFile] -> <@@ (%%csvFile:CsvFile).Data @@>))

ty.AddMember(rowTy)

ty)

do this.AddNamespace(ns, [csvTy])

Here are a few additional points to note about the implementation:

* Overloaded constructors allow either the original file or one with an identical schema to be read. This is a common pattern when writing type providers for local or remote data sources, and allows a local file to be used as the template for remote data.
* We use the AddDefinitionLocation method to define the location of the provided properties. This means that using “Go To Definition” on a provided property will open the CSV file in Visual Studio.
* We use the ProvidedMeasureBuilder type to look up the SI units and to generate the relevant float<\_> types.

Key Lessons

In this section you have learned how to create a type provider for a local data source with a simple schema contained in the data source itself.

# Going Further

## Design and Naming Conventions for Type Providers

### Providers for Connectivity Protocols

In general most provider DLLs for data and service connectivity protocols like OData or SQL connections should end in “TypeProvider” or “TypeProviders”. For example, use a DLL name like this:

#r "Fabrikom.Management.BasicTypeProviders.dll"

And ensure that your provided types live under the corresponding namespace, and indicate the connectivity protocol implemented:

Fabrikom.Management.BasicTypeProvider.WmiConnection<…>

Fabrikom.Management.BasicTypeProvider.DataProtocolConnection<…>

### Utility Providers for General Coding

For a utility type provider such as that for regular expressions, the type provider may be part of a base library, e.g.:

#r "Fabrikom.Core.Text.Utilities.dll"

In this case the provided type would appear at an appropriate point according to normal .NET design conventions:

open Fabrikom.Core.Text.TypeCheckedRegEx

let regex = RegEx<"a+b+a+b+">

### Singleton Data Sources

Some type providers connect to a single dedicated data source and just provide data. In this case, the “TypeProvider” is best dropped, and normal .NET naming conventions used:

#r "Fabrikom.Data.Freebase.dll"

let data = Fabrikom.Data.Freebase.Astronomy.Asteroids

See also the “GetConnection” design convention below.

## The GetConnection Design Convention

Most type providers should be written to use the “GetConnection” pattern used by the type providers in **FSharp.Data.TypeProviders.dll**, e.g.

#r "Fabrikom.Data.WebDataStore.dll"

type Service = Fabrikom.Data.WebDataStore<…static connection parameters…>

let connection = Service.GetConnection(…dynamic connection parameters…)

connection.Astronomy.Asteroids

## Creating Types and Members On-demand

The ProvidedType API has “Delayed” versions of AddMember.

type ProvidedType =

member AddMemberDelayed : (unit -> MemberInfo) -> unit

member AddMembersDelayed : (unit -> MemberInfo list) -> unit

These are used to create “on-demand” spaces of types.

## Adding Static Parameters

Provided types may have static value parameters. An example could be, CSVFile:

CSVFile<”schema.txt”>

Here CSVFile is not a type itself, but is rather a provided type with a static parameter. When it is applied it yields a type. This is like generic types, except the parameters are values and the instantiated type shape can depend on the value of that parameter.

You add static parameters to a type by using DefineStaticParameters

t.DefineStaticParameters(parameters = [ ],

instantiationFunction=(fun typeName args -> ...))

The instantiationFunction must return the provided type, which should have the given typeName. The arguments passed to the instantiationFunction are objects, one for each static parameter.

## Providing Array, ByRef and Pointer types

You make provide members whose signatures include array types, byref types and instantiations of generic types by using the normal MakeArrayType, MakePointerType and MakeGenericType on any instance of System.Type, including ProvidedTypeDefinitions.

## Providing Unit of Measure Annotations

The ProvidedTypes API provides helpers for providing measure annotations. For example, to provide the type double<kg> use:

let measures = ProvidedMeasureBuilder.Default

let kg = measures.SI "Kilogram"

let m = measures.SI "Meter"

let float\_kg = measures.AnnotateType(typeof<double>,[kg])

To provide the type Nullable<decimal<kg/m^2>> use:

let kgpm2 = measures.Ratio(kg, measures.Square m)

let dkgpm2 = measures.AnnotateType(typeof<decimal>,[kgpm2])

let nullableDecimal\_kgpm2 = typedefof<System.Nullable<\_>>.MakeGenericType [|dkgpm2 |]

## Accessing Project-Local or Script-Local Resources

Each instance of a type provider can be given an TypeProviderConfig. This contains the “resolution folder” for the provider, i.e. the project folder for the compilation or the directory containing a script, as well as the list of referenced assemblies and other information.

## Invalidation

Providers can raise invalidation signals to notify the F# Language Service that the schema assumptions may have changed. A re-typecheck will occur if the provider is being hosted in Visual Studio.

## Caching Schema Information

Providers will often need to cache access to schema information. The cached data should be stored using a file name given as a static parameter, or as user data.

## Type Providers Backed By Remote Data and Services

Type providers backed by remote data and services must consider a range of issues inherent in connected programming. These include

* Schema mapping
* Liveness and invalidation in the presence of schema change
* Schema caching
* Async implementations of data access operations
* Supporting queries, including LINQ queries
* Credentials & Authentication

These issues are not explored in depth in this article beyond the notes above.

## 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, the host compiler attributes the error to a specific type provider. Type provider exceptions should never result in internal compiler errors.

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

## Some Rules For Type Providers

### Provided types must be reachable

All ProvidedTypes should be reachable from the non-nested types. The non-nested types are given in the array to ProvidedTypeProvider(). For example, if you have “StaticClass.P : T” you must ensure T is either one of the non-nested types or nested under one.

For example, some providers have a static class like “DataTypes” containing these T1,T2,T3,... types. If you forget this, the error says:

“A reference to type T in assembly A was found, but the type could not be found in that assembly”

If you see this, check that all your subtypes can be reached from the provider types. Note: These T1, T2, T3... are referred to as the “on-the-fly” types. Remember to put them somewhere.

## Development Tips

### Run Two Visual Studio Instances

You can use one Visual Studio to develop the type provider. You can test it in another, since the test IDE will take a lock on the DLL preventing it being rebuilt. Thus, the second instance of Visual Studio must be closed while the provider is built in the first instance, and then reopened.

### Debug type providers by using invocations of fsc.exe

Type providers are invoked by

* fsc.exe (The F# Command Line compiler)
* fsi.exe (The F# Interactive compiler)
* devenv.exe (Visual Studio)

Debugging type providers can often be easiest using fsc.exe on a test script file (e.g. **script.fsx**). This is because

* You can launch debug using a command-line prompt

devenv /debugexe fsc.exe script.fsx

* You can use print-to-stdout logging

## Some limitations of the F# Type Provider Mechanism

The following limitations apply to the F# Type Provider mechanism:

1. Provided generic types are not supported by the underlying F# type provider infrastructure
2. Provided generic methods are not supported by the underlying F# type provider infrastructure
3. Nested types with static parameters are not supported (again, this is a limitation of the underlying infrastructure).

## Some limitations of the ProvidedTypes support code

The following limitations apply to the ProvidedTypes support code:

1. Provided properties with indexed getters and setters are not yet implemented.
2. Provided events are not yet implemented.
3. The provided types and info objects should only be used for the F# type provider mechanism. They are not more generally usable as System.Type objects
4. There are several limitations on the constructs that can be used in quotations defining method implementations.