Strongly-Typed Language Support for Internet-Scale Information Sources

Don Syme, Keith Battocchi, Kenji Takeda[1], Donna Malayeri, Jomo Fisher, Jack Hu, Tao Liu, Brian McNamara, Daniel Quirk, Matteo Taveggia, Wonseok Chae, Uladzimir Matsveyeu[2], Tomas Petricek[3]

[1] Microsoft Research, Cambridge, United Kingdom

[2] Microsoft Corporation, Redmond WA, USA

[3] University of Cambridge, United Kingdom

Abstract. A growing trend in both the theory and practice of programming is the interaction between programming and rich information spaces. From databases to web services to the semantic web to cloud-based data, the need to integrate programming with heterogeneous, connected, richly structured, streaming and evolving information sources is ever-increasing. Most modern applications incorporate one or more external information sources as integral components. Providing strongly typed access to these sources is a key consideration for strongly-typed programming languages, to insure low impedance mismatch in information access. At this scale, information integration strategies based on library design and code generation are manual, clumsy, and do not handle the internet-scale information sources now encountered in enterprise, web and cloud environments. In this report we describe the design and implementation of the *type provider* mechanism in F# 3.0 and its applications to typed programming with web ontologies, web-services, systems management information, database mappings, data markets, content management systems, economic data and hosted scripting. Type soundness becomes relative to the soundness of the type providers and the schema change in information sources, but the role of types in information-rich programming tasks is massively expanded, especially through tooling that benefits from rich types in explorative programming.

1 Introduction

A key direction for the future evolution of programming is to allow strongly typed programming to "escape the box" of type structures defined in hand-written or tool-generated code, and to systematically bridge the gap between the language and the schematized information found in external information systems. In this report

- We describe the design and implementation of a novel type-bridging mechanism, the *type provider* mechanism in F# 3.0.
- We describe its applications to strongly typed programming with web ontologies, web-services, database mappings, directory navigation, content management systems, scientific data sets and hosted scripting.
- We consider the tradeoffs of these mechanisms, including the relative soundness properties of the different systems that may be designed and implemented.
- We describe how type-bridging both radically expands the role for names and types, but also challenges existing, comfortable assumptions about what types are, how they are selected and what properties they should have.
- We illustrate the relative ease-of-use of the type provider mechanism as compared to alternate technologies, in addition to its performance and scaling benefits.

While we have made valuable initial progress for supporting information-rich applications, we believe that this area is an excellent opportunity for future language and tooling research, information-space modeling, schematization techniques, and language usability efforts.

This report is structured as follows. In Section 2, we consider the problem of information-rich programming, especially in the context of strongly-typed languages. Section 3 presents the type provider mechanism and explains its role in addressing information-rich programming problems, and Section 4 looks at specific examples of using the mechanism to integrate "internet-scale" information sources. Section 5 looks at themes that arise when using the type provider mechanism in practice, many of which raise interesting future R&D directions. In Section 6, we briefly describe how information-rich programming can affect our view of the logical characteristics usually associated with programming languages such as type-soundness. In Section 7 we describe other applications we have explored with the type provider mechanism, and in Section 8 we summarize, describe related work and future directions.

2 The Problem

As most of us know and experience every day, the world is now enormously information rich. It is now common wisdom that we are witnessing an explosion of digital data and information. Further, that information is increasingly available through high reliability, well-organized, curated services. For example, Figure 1 shows the rise in availability of "open APIs", i.e. web-based APIs delivering digital information and services, as recorded on the catalog programmableweb.com.

Programmed services and applications can now be viewed as components that consume, filter, transform and re-republish information within a larger connected and reactive system. Modern applications (both enterprise and "apps") increasingly integrate one or more external information sources as an integral component within the application.

Despite this, few strongly-typed programming languages and tools are able to seamlessly integrate external information sources as if they were strongly-typed components from a programmer's perspective. We believe that in coming years, we will continue to see more and more applications that are information-focused (as opposed to code-focused), and as a result this is an area of programming language design that requires more attention.

Practically, speaking, interacting with external information systems from strongly-typed programming languages has reached an impasse.

The size and number of information spaces is growing rapidly, with respect to both data and metadata. Stable, organized information spaces of enormous size are now available through networked services (and thus for use

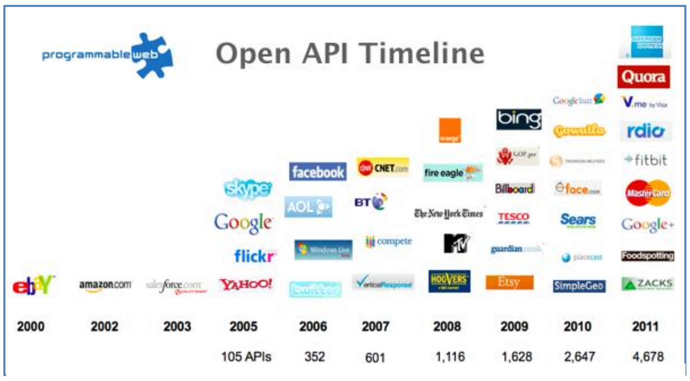


Figure 1 - The Growth of the Open Digital Information Context, with API counts

from programming languages). Importantly, these spaces are both huge in terms of absolute amounts of data (e.g. total number of data points or tuples), and in absolute amounts of metadata (e.g. total size of organized schemas, names and documentation associated with the data).

There will never be a single universal schema language or protocol. Schematization and protocols for organized information are a rapidly developing milieu of overlapping technologies and standards. Our natural instinct as computer scientists is to seek a single, unifying standard language for describing schematized data sources. The history of software is littered with such attempts: SQL, XML, Web Services, CORBA, COM, Linked Data, OData, GData, Atom, REST, RSS, JSON, RDF, schema.org: the list is daunting and always growing. Both the "data format wars" and "protocol wars" show no sign of ceasing, and are often driven by commercial interests and economic network effects resistant to arguments based on technical merits. As a result technologies often trend towards lowest-common-denominator approaches. Our programming languages must rise above this milieu and adopt information integration architectures that are open, rather than tied to particular standards.

Traditional typed bridging mechanisms don't scale. Three techniques are traditionally used to bridge programming languages to information sources:

- hand-written static libraries,
- generated static libraries,
- dynamically-typed information representation.

We discuss specific examples of these techniques as related work in Section 8. In short, hand-written libraries do not scale to information spaces with large metadata-size, e.g. with hundreds or thousands of different "types" in the information space. Generated static libraries scale better, but have other problems: workflows involving code generation are clumsy and do not integrate well with explorative programming; schemas are read eagerly instead of on-demand; code-bloat can arise; the generated code can be fragile; and the technique generally doesn't scale to information spaces with thousands or millions of types.

For instance, one might wish to build an application that uses data from Freebase [BEP+08], a Creative Commons-licensed repository containing nearly 22 million structured data entities—ranging from the periodic table (e.g., the atomic mass of hydrogen), to information on Hollywood celebrities (e.g. their legal entanglements and rehab facilities). Clearly, it would be impossible to use code generation techniques to access Freebase's schema and metadata: the amount of generated code would just be too big. Due to the interconnectedness of the entity graph, trying to generate types for just a small subset can still end up pulling in the entire graph.

Likewise, consider the problem of language-integration of the Azure Data Market [Mic12b], an online directory and hosting service containing hundreds of information sources, all of them schematized to some degree, and each with a Service-level agreement (SLA) to provide schema stability. Using code-generation to access this data store is unappealing, and using dynamic information representation techniques clearly sub-optimal except where using a typed language is not an option.

We expect that in the future we will continue to see more data sources like Freebase and Azure Data Market, where the "internet-scale" of these information services alone makes code generation unworkable. Further examples are considered in Section 4.

Dynamically-typed bridging mechanisms discard the benefits of strongly-typed programming. Using dynamically-typed information representation scales well but is a last-resort that discards the benefits of strongly-typed programming. The use of dynamic representation techniques is particularly disturbing when working against schematized information sources that come equipped with fully stable, high-value schemas – in this situation there seems no reason, *per se*, why strong typing should not be applicable. It also loses the performance, tooling, correctness and cross-component interoperability benefits associated with strong types. However, if strongly-typed languages don't have any understanding of the schemas of the external data that programmers are actually using then a strongly-typed language becomes an ever-less-appealing option. Strongly-typed languages are left out in the cold, even when they contain expressive mechanisms for manipulating structured and remote data.

2.1 Some Definitions

Before we go further, we offer some definitions to aid discussion.

An *information space* is a loose notion that captures data sources, external to the programming language, optionally annotated with meta-data. Information spaces include SQL databases (with database schema as meta-data), XML files (without meta-data or with XSD schema), semantic web with rich meta-data or unstructured data.

Information-rich programming is programming where one or more information spaces are integral to the operation of the programs being constructed. These may be as simple as textual DSLs embedded as strings in the program itself, as familiar as SQL databases, or as massive as a service exposing Wikipedia data, or the world-wide-web of HTML documents itself.

An *information space schema* is a (often formal) structure that characterizes the common names, shapes, operations and constraints for an external information space. For some information spaces, schematization has been performed as an integral part of the design methodology of the information space. In others, schematization is more arbitrary and the optimal schematization depends on the pattern of use. The schematization is chosen to minimize the complexity of accurately working with the residual information. A schematization does not typically fully define the data space (e.g. does not contain specific leaf values).

A *(strongly-typed) information-rich programming language* is a language that allows the integration of external information sources, where the schema and content of these sources are presented in a (strongly-typed) idiomatic form. Such idiomatic must reflect the *information space schema* of the information space in the strongly-typed representation on the programming language side.[1]

A *component signature* is the signature of software component or information space when considered as a component in the host programming language. The signature typically will contain types, methods and properties and additional metadata such as documentation.

A *type-bridging technique* is a mechanism and/or methodology to take specified information spaces and produce programming language projections of those, including both a component signature and a component implementation.

[1] For example, an object-oriented language which presents all external data acquired through an HTTP connection as strings would not be considered information-rich, since the information is not presented in idiomatic form.

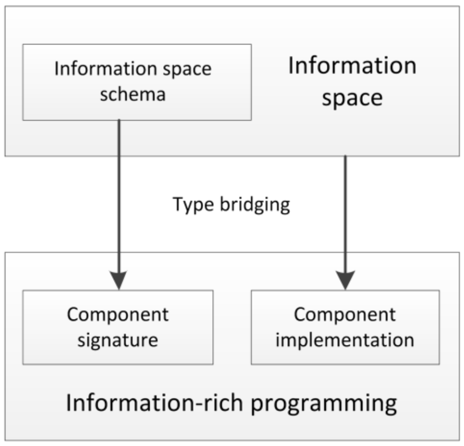


Figure 2 – Core relationships in information-rich programming.

A *language integrated query mechanism* is a way of writing queries in the host programming language which are then passed to external information sources. This frequently involves authoring the queries using some form of meta-programming.

3 The Technique: Type Providers

We now describe the novel language and tooling construct we have implemented to help address the problem of integrating internet-scale information services into F# 3.0. We call this mechanism *F# 3.0 type providers*, or just *type providers* for short.

A type provider is a compile-time component that, given optional static parameters identifying an external information space and a way of accessing that information space, provides two things to the host F# compiler/tooling:

(1) A *provided component signature* that acts as the programming interface to that information space, and which is computed on-demand as needed by the F# compiler. For F#, the component signature contains provided namespaces, types, methods, properties, events, attributes, and literals that give a .NET object-oriented characterization of the information space.

(2) A *provided component implementation* of the component signature. This is given by either an actual .NET assembly that implements the component signature (the *generative* model for the provided types), or, a pair of erasure functions giving *representation types* and *representation*

expressions for the provided types and provided methods respectively (the *erasure* model for the provided types).

Put simply, type providers are about using a provider model for the "type import" logic of the host language compiler or tooling. Essentially, a type provider is an adapter component that reads schematized data and services and transforms them into types in the target programming language. This allows programmers to quickly leverage rich, schematized information sources without an explicit transcription process (be it code generation or a manually created ontology). The provided types can then be leveraged by not only the type-checker and runtime, but also tools that rely on the type-checker, such as IDE auto-completion. Additionally, if the data source contains additional descriptive metadata (such a description of various columns in a database), this can be transformed by the type provider into information that is visible to the programmer within the IDE (such as documentation contained in tooltips).

A type provider does not necessarily contain any types itself; rather, it is a component for generating descriptions of types, methods and their implementations. A type provider is thus a form of compile-time meta-programming, a compiler plugin with access to the external world that augments the set of types that are known to the type-checker and compiler.

Importantly, a type provider provides types and methods *on-demand*, i.e. lazily, as the information is required by the host tool such as the F# compiler. This allows the provided type space to be very large or even infinite. We consider this further in Section 6.

As mentioned above, the implementation of a provided component is given either by generation or by erasure. If the erasure is used, then it is described through a *type erasure function* mapping provided types to representations (in F#, this is the first non-erased super type of the type), and a *member erasure function* that gives an expression which replaces each use of a provided member (see GetInvokerExpression in Appendix A). When using erasure, unnecessary code bloat in the corresponding compilation artifact is avoided. When using type-generation, existing code generation tools can be wrapped and presented as type providers, and exact .NET runtime type information is preserved.

A mini-formalization of a calculus related to type providers is described in Section 6. The low-level API for a type provider is described in Appendix A.

An example of implementing a type provider using a higher-level API is described in Appendix B.

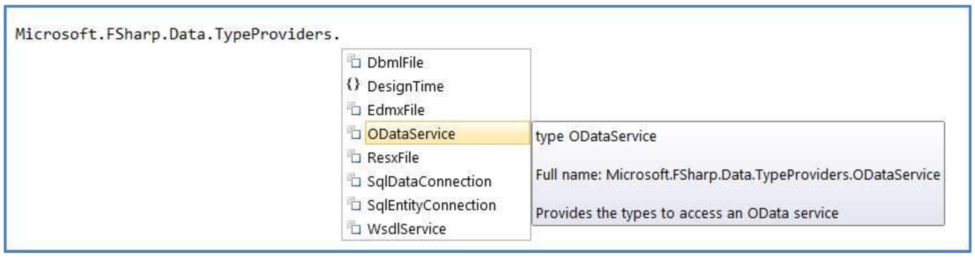
4 Integrating Internet-Scale Information Services

4.1 Example: OData

We now describe the use of F# 3.0 type providers to integrate an example internet data protocol into F# programming. We use OData [COP12] as our example. OData is a protocol for querying data sources over HTTP and is ultimately implemented by REST requests. In traditional use, a programmer uses a code generator to access a particular service within her programming language of choice. Alternatively, accessing an OData service requires manually building up URLs with embedded strings to represent queries and retrieving the response as text. The former is clumsy, the latter is both error-prone and gives an untyped view of the data.

```
open Microsoft.FSharp.Data.TypeProviders
type NetFlix = ODataService<"http://odata.netflix.com/Catalog/">
let netflix = NetFlix.GetDataContext()

let avatarTitles =
   query { for t in netflix.Titles do
           where (t.Name.Contains "Avatar")
           sortBy t.Name
           take 100 }
```

Figure 3. Accessing the NetFlix data source using the OData type provider

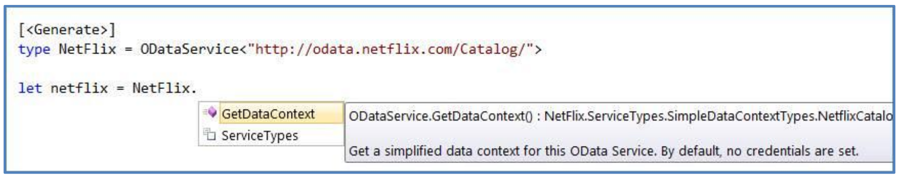

```
Microsoft.FSharp.Data.TypeProviders.
```

type ODataService

Full name: Microsoft.FSharp.Data.TypeProviders.ODataService

Provides the types to access an OData service

Figure 4 - Choosing the OData type provider



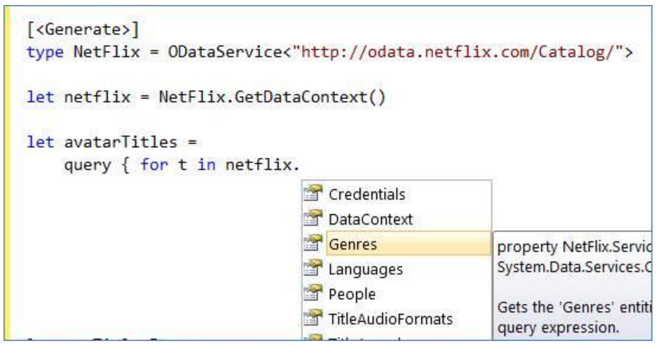
```
[<Generate>]
type NetFlix = ODataService<"http://odata.netflix.com/Catalog/">

let netflix = NetFlix.
```

GetDataContext
ServiceTypes

ODataService.GetDataContext() : NetFlix.ServiceTypes.SimpleDataContextTypes.NetflixCatalo

Get a simplified data context for this OData Service. By default, no credentials are set.

Figure 5 - Getting a Data Context



```
[<Generate>]
type NetFlix = ODataService<"http://odata.netflix.com/Catalog/">

let netflix = NetFlix.GetDataContext()

let avatarTitles =
    query { for t in netflix.
```

Credentials
DataContext
Genres
Languages
People
TitleAudioFormats

property NetFlix.Servic
System.Data.Services.C

Gets the 'Genres' entiti
query expression.

Figure 6 - Exploring the Information Service

In contrast, F# 3.0 includes an "in-the-box" implementation of an OData type provider, which utilizes the built-in .NET code generator for OData and side-steps these issues. A programmer uses this type provider as follows:

- Reference the OData type provider, similar to any assembly reference (Figure 3, line 1). Auto-completion helps the programmer select the correct type provider (Figure 4).
- Fill in the parameters for the connection (Figure 3, line 5).
- Get a data context (line 7), which acts as an object to encapsulate additional runtime parameters such as credentials. Figure 5 shows auto-completion.
- Write a query against the data source (lines 9–13). Figure 6 shows the programmer exploring the provided services with the help of strong-typing and auto-completion.

The alternative would be to use a code generation approach, which does not provide such a clean integration with the user's scripting and programming environment.

The Provided Types and Representation Functions

Under the hood, the F# 3.0 OData provider generates a type space for the schema using the "datasvcutil.exe" code generator available in the .NET Framework, augmented with a thin veneer of "wrapper" types to simplify the view of the presented information space. This is an example where an existing code generator is used in the type provider framework. Because the provided assembly is generated, the type representation function is a 1:1 correspondence with generated .NET types.

The main advantage that F# type providers brings is that it allows a totally code-focused and scripting-friendly programming experience of the data (which contrasts with the need to switch to another application, manually invoke an external tool and import the generated code). For example, multiple OData sources can be mashed up within a single script file.

4.2 Example: Freebase

Traditional web services can feature reasonably large metadata descriptions involving hundreds of types. However, in practice, data sources and data directories are now appearing that feature much larger quantities of metadata and types. Importantly, the F# 3.0 type provider mechanism can scale to these services. To illustrate this, we discuss a type provider that integrates the entity graph Freebase [BEP+08] into F# in a strongly-typed way.

Freebase is self-described as "an entity graph of people, places and things, built by a community that loves open data". It contains a great deal of interesting and useful structured data highly suited for integration into programming applications. Much of the information is drawn from open sources such as Wikipedia.

Consider the problem of constructing a strongly-typed API for the Chemical Elements using the data and schema from www.freebase.com. The core chemistry schema in Freebase in question has 10s of types (*Elements*, *Isotopes*, *People who Discovered Elements*, *Discovering Countries*, …), each with 100s of methods and properties (*Atomic Number*, *Atomic Radius*, …). It is, in fact, embedded and linked to other entities in the much larger overall schema. At the time of writing, Freebase featured 23,000 types, 61,000 properties and millions of entities. Using traditional library-authoring techniques, a subset of this API would take weeks to isolate, construct, document, populate with data and test.

This begs the question: why can't we use this web database *directly*, as if it were "part of our program"? After all, Freebase is well-schematized, and that schema appears to be quite stable for most practical purposes. It also provides a REST service endpoint to query both data and schema through a bespoke query language called MQL, which can be exercised directly via HTTP GET operations supported in all modern languages, returning data in a JSON format. However, using these results is awkward, especially when using a strongly typed language, and error-prone. Using the REST service requires considerable skills and manual coding.

From the traditional programming languages perspective, Freebase is, like many information sources, an example of an external information source that happens to have a "type-like" system. Faced with the above task in a traditional strongly-typed language, programmers would be forced to either eschew strong typing, or turn to either meta-programming or code generation.

- In the former case, the programmer manually writes strongly-typed record (or class) types to represent the information space. If the language supports a form of code annotations (.NET attributes or Java annotations), they can be used to indicate how these types correspond to the external information space. The programmer then constructs a meta-model mapping tool that correlates these annotations with the external information space [SGC07].
- When using code generation, they would write a tool which automatically generates API code based on the Freebase schema. However, in this case the API designer would still have to carve out a "boundary" of the information source, since the generated API would otherwise be too big.

This leads to an impasse which is impossible to solve in traditional approaches to strongly typed language design and implementation.

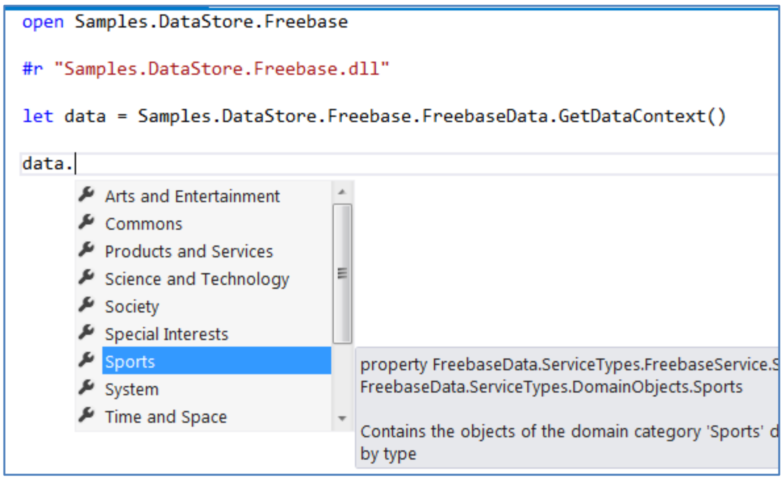
Instead, consider how we address this problem with an F# 3.0 type provider specifically designed for Freebase. First we describe how the information space appears to the F# programmer when using this type provider. From a script, the F# 3.0 type provider is referenced in the same way as a library:

```
#r "Samples.DataStore.Freebase.dll"
```

Likewise, you can reference the library in a command-line invocation:

```
fsc -r:Samples.DataStore.Freebase.dll ChemistryProgram.fs
```

Once referenced, the provided data space can be explored by first getting a data context and then using auto-completion, see Figure 7.



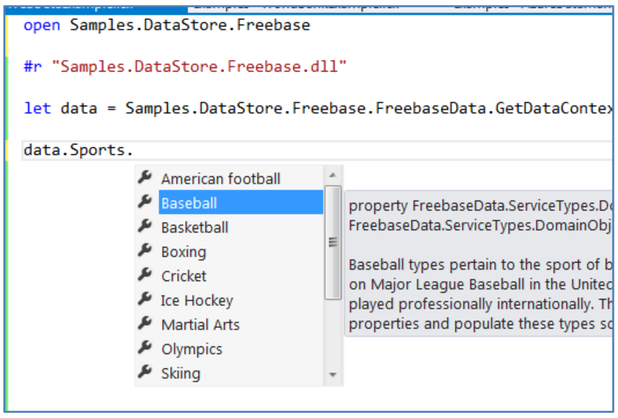
```
open Samples.DataStore.Freebase

#r "Samples.DataStore.Freebase.dll"

let data = Samples.DataStore.Freebase.FreebaseData.GetDataContext()

data.
```

property FreebaseData.ServiceTypes.FreebaseService.S
FreebaseData.ServiceTypes.DomainObjects.Sports

Contains the objects of the domain category 'Sports' d
by type

Figure 7 - Exploring Freebase with the F# Freebase Type Provider

Once a domain category such as “Sports” is selected, the individual domains can now be examined:



```
open Samples.DataStore.Freebase

#r "Samples.DataStore.Freebase.dll"

let data = Samples.DataStore.Freebase.FreebaseData.GetDataContex

data.Sports.
```

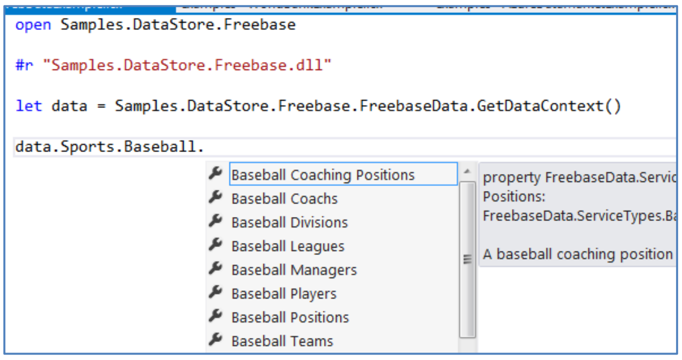
- American football
- Baseball
- Basketball
- Boxing
- Cricket
- Ice Hockey
- Martial Arts
- Olympics
- Skiing

property FreebaseData.ServiceTypes.D
FreebaseData.ServiceTypes.DomainObj

Baseball types pertain to the sport of b
on Major League Baseball in the Unitec
played professionally internationally. Th
properties and populate these types sc

Figure 8 - Exploring the Sports domain category in Freebase

For a given domain such as Sports.Baseball, queryable collections of objects for all individual types associated with the domain are then shown:



```
open Samples.DataStore.Freebase

#r "Samples.DataStore.Freebase.dll"

let data = Samples.DataStore.Freebase.FreebaseData.GetDataContext()

data.Sports.Baseball.
```

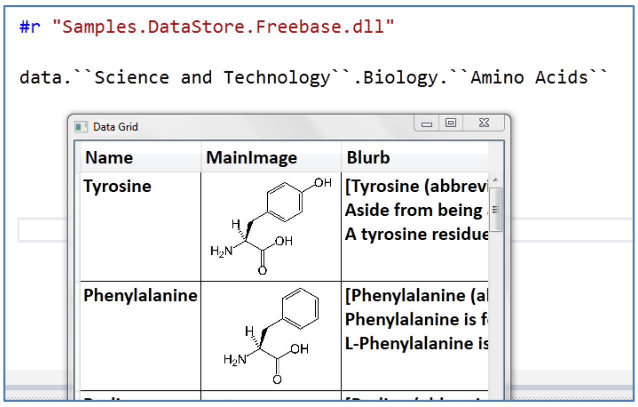
- Baseball Coaching Positions
- Baseball Coachs
- Baseball Divisions
- Baseball Leagues
- Baseball Managers
- Baseball Players
- Baseball Positions
- Baseball Teams

property FreebaseData.Servic
Positions:
FreebaseData.ServiceTypes.B

A baseball coaching position

Figure 9 - Exploring the Baseball Domain on Freebase

The data sets (organized by type) can now be visualized, for example the Amino Acids



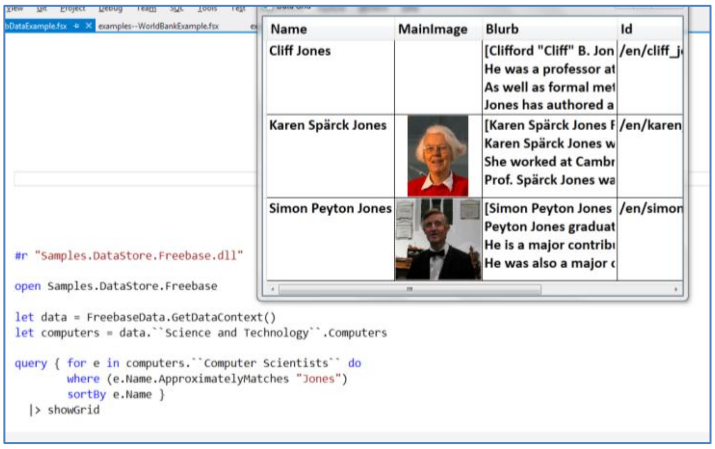
```
#r "Samples.DataStore.Freebase.dll"

data.``Science and Technology``.Biology.``Amino Acids``
```

Data Grid

Name	MainImage	Blurb
Tyrosine		[Tyrosine (abbrev Aside from being A tyrosine residue
Phenylalanine		[Phenylalanine (al Phenylalanine is f L-Phenylalanine is

Figure 10 - Visualizing a Data Set Selected by Freebase Type

Finally, data sets can be enumerated, compositionally analyzed using functional programming combinators, or queried using LINQ queries. Visualizing the results of the queries is shown in Figure 11.



Name	MainImage	Blurb	Id
Cliff Jones		[Clifford "Cliff" B. Jon He was a professor at As well as formal met Jones has authored a	/en/cliff_j
Karen Spärck Jones		[Karen Spärck Jones F Karen Spärck Jones w She worked at Cambr Prof. Spärck Jones wa	/en/karen
Simon Peyton Jones		[Simon Peyton Jones Peyton Jones graduat He is a major contrib He was also a major c	/en/simon

```
#r "Samples.DataStore.Freebase.dll"

open Samples.DataStore.Freebase

let data = FreebaseData.GetDataContext()
let computers = data.``Science and Technology``.Computers

query { for e in computers.``Computer Scientists`` do
        where (e.Name.ApproximatelyMatches "Jones")
        sortBy e.Name }
  |> showGrid
```

Figure 11 - Querying Computer Scientists with the Freebase Type Provider

The Provided Types and Representation Functions

The provider component is used by the F# compiler to resolve type names in the F# source code to provided types and members. The provider resolves the name "Samples.DataStore.Freebase" (abbreviated to S.F.D) to the following information:

```
Samples.DataStore.Freebase →
  // ...maps to a provided space of names containing...

  static member GetDataContext : unit -> S.F.D.FreebaseContext
```

and further names such as “S.F.D.FreebaseContext” resolve as follows:

```
  S.F.D.FreebaseContext →
    // ...maps to a provided type containing the property...
    ``Science and Technology`` : S.F.D.Categories.``Science and Technology``

  S.F.D.Categories.``Science and Technology`` →

     // ...maps to a provided type containing the property...
     Chemistry : S.F.D.Domains.Chemistry

  S.F.D.Domains.Chemistry →
      // ...maps to a provided type containing the property...
      ChemicalElements : seq<S.F.D.Types.ChemicalElement>

  S.F.D.Types.ChemicalElement →
      // ...maps to a provided type corresponding to the Freebase
      // type /type/chemistry/chemical_element containing the properties:
      Name : string
      AtomicNumber : string
      Istotopes : seq<S.F.D.Types.ChemicalIsotope>

  S.F.D.Types.ChemicalIsotope →

      // ...maps to provided type corresponding to the Freebase
      // type /type/chemistry/isotope containing the properties:
      Name : string
      Element : ChemicalElement
```

This information is automatically computed from the schema information provided by the Freebase metadata service API, on-demand, as needed by the F# compiler. The following program then typechecks:

```
let main() =
  let ctxt = Samples.DataStore.Freebase.GetDataContext()
  let chemistry = ctxt.``Science and Technology``.Chemistry.ChemicalElements
  for elem in elements do
    printfn "element %s has %d isotopes" elem.Name (elem.Isotopes.Count())
```

As described in Section 3, the Freebase type provider specifies a *type representation function* for the provided types. This is very simple: provided types map to library types FreebaseContext and FreebaseObject, for example:

```
  Samples.DataStore.Freebase        → FreebaseContext
  Samples.DataStore.Freebase.``Science and Technology``
        .Chemistry.ChemicalElement   → FreebaseObject
        .Chemistry.ChemicalIsotope   → FreebaseObject
```

The provider also specifies a *member representation function* for the operations on these types. This is also simple – accessing a collection of all objects of a particular type maps to a call to a library helper method

GetObjects, passing the unique Freebase identifier for the type, and all property accesses maps to a library helper method GetProperty, again passing the unique Freebase identifier for the property:

```
Samples.DataStore.Freebase.GetDataContext()
      → new FreebaseContext()

ctxt.``Science and Technology``.Chemistry.ChemicalElements
      → ctxt.GetObjects("/chemistry/chemical_element")

elem.``Atomic Number``
      → elem.GetProperty("/chemistry/chemical_element/atomic_number")

elem.Isotopes
      → elem.GetProperty("/chemistry/chemical_element/isotope")

isotope.``Half-life``
      → isotope.GetProperty("/chemistry/isotope/half_life")

isotope.``Isotope of``
      → isotope.GetProperty("/chemistry/isotope/isotope_of")
```

GetObjects and GetProperty communicate with the Freebase service using HTTP, extracting the requested data.

The program above then compiles to the equivalent of the following once the given representation functions are applied:

```
  let main() =
    let ctxt = new FreebaseContext()
    let elements = ctxt.GetObjects("/chemistry/chemical_element")
    for elem in elements do
      printfn "element %s has %d isotopes"
       (elem.GetProperty("/chemistry/chemical_element/atomic_number"))
       (elem.GetProperty("/chemistry/chemical_element/isotope").Count())
```

All the provided types and operations have been erased.

In practice, the experimental Freebase provider implemented by the F# team differs from the outline given here in a number of ways:

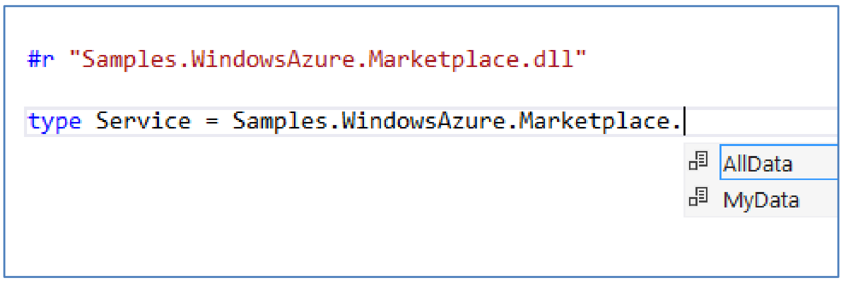
- Freebase types are mapped to .NET interface types that support multiple inheritance.
- Freebase objects are represented as property bags.
- Properties may be empty, so the .NET Nullable types are used to represent potential absence of information.
- Numeric data is, where possible, given a unit-of-measure based on the unit metadata available in Freebase. The details of unit-of-measure projection are beyond the scope of this report.

- The collections returned support LINQ IQueryable [MBB06], meaning composite query operations built from these collections are translated into Freebase MQL queries. The details of this query translation are beyond the scope of this report.

4.3 Example: Azure Data Market

The Windows Azure Marketplace is a web-hosted directory and store for applications, data and information services. At the time of writing the data sets include 170 data sets (60 free, 80 paid and 30 free trial) in areas from UN and World Bank data sets to Sports information databases, with more being added weekly. Many are very large, with trillions of data points in total, and all are suitable for implementing data mashups, web applications or as backing data for mobile application development. Users have an account with the data market and subscribe to data sets (subscription is required for free data sets). Paid data sets are billed based on the transaction volumes. The data subscriptions come with SLA guarantees, for example some guarantee that the schema of the data source will be stable for 1 year. Schemas for all data sets are available without subscription or log-on.

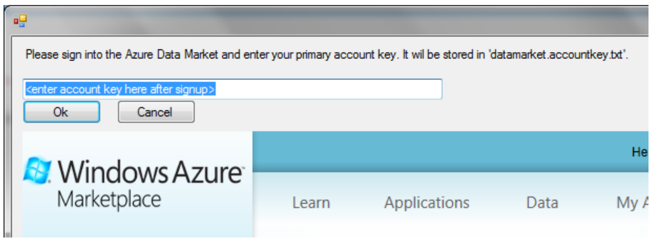
We have created a prototype F# type provider that embeds the entire data market within the name and type space of F#. Figure 12 shows the use of the initial navigation of the data market, revealing that the data market embedding includes both “all” data sets (all data sets with their schemas) and “my” data sets (ones the user is subscribed to).



```
#r "Samples.WindowsAzure.Marketplace.dll"

type Service = Samples.WindowsAzure.Marketplace.
```

Figure 12 - Navigating the Data Market

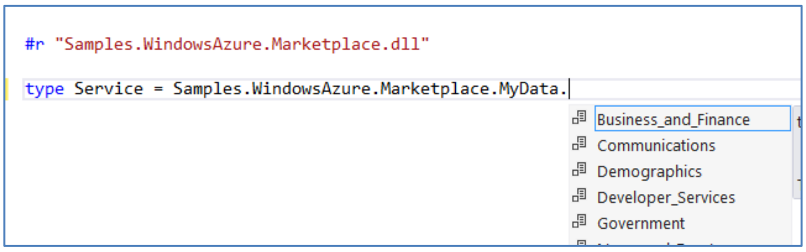
When the user first uses “MyData” types in an IDE environment, the user is prompted to create a data market account and/or log on to that account. Figure 13 shows a dialog for this. An access token for the account is then stored. (If using the F# command-line compiler or the F# REPL, the stored access token is used).



Please sign into the Azure Data Market and enter your primary account key. It will be stored in 'datamarket.accountkey.txt'.

Figure 13 - Signing into the Data Market when accessing "MyData" from IDE

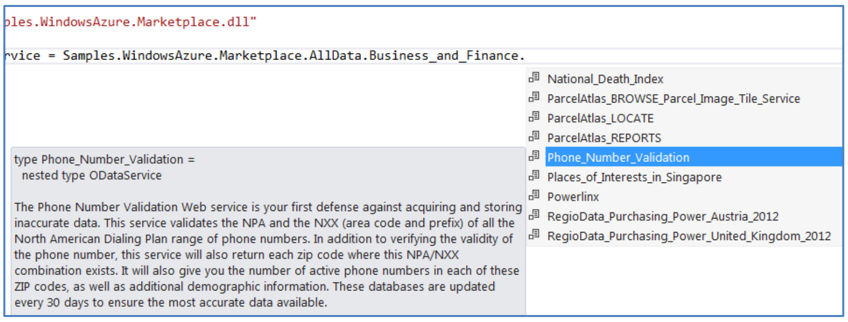
After sign-in, the data sets we are subscribed to are under MyData, shown in Figure 14.



```
#r "Samples.WindowsAzure.Marketplace.dll"

type Service = Samples.WindowsAzure.Marketplace.MyData.
```

Figure 14 - Subscribed data sests

If the user navigates “AllData”, then all data sets are shown, see Figure 15.



```
les.WindowsAzure.Marketplace.dll"

vice = Samples.WindowsAzure.Marketplace.AllData.Business_and_Finance.
```

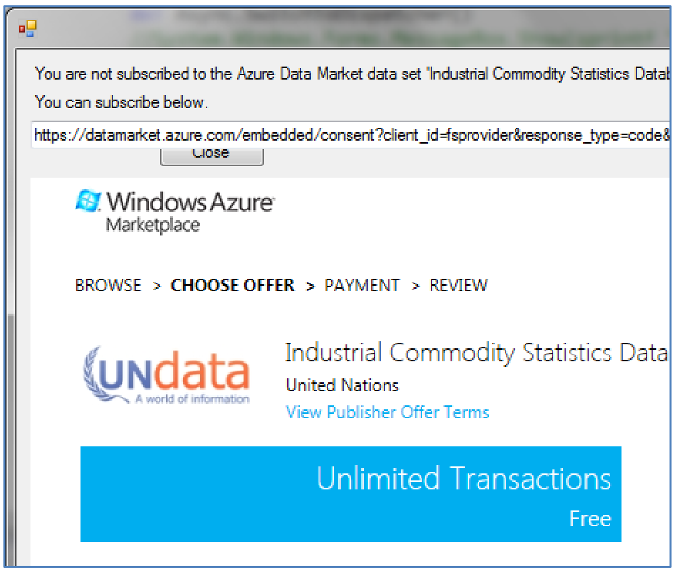
National_Death_Index
ParcelAtlas_BROWSE_Parcel_Image_Tile_Service
ParcelAtlas_LOCATE
ParcelAtlas_REPORTS
Phone_Number_Validation
Places_of_Interests_in_Singapore
Powerlinx
RegioData_Purchasing_Power_Austria_2012
RegioData_Purchasing_Power_United_Kingdom_2012

type Phone_Number_Validation =
  nested type ODataService

The Phone Number Validation Web service is your first defense against acquiring and storing inaccurate data. This service validates the NPA and the NXX (area code and prefix) of all the North American Dialing Plan range of phone numbers. In addition to verifying the validity of the phone number, this service will also return each zip code where this NPA/NXX combination exists. It will also give you the number of active phone numbers in each of these ZIP codes, as well as additional demographic information. These databases are updated every 30 days to ensure the most accurate data available.

Figure 15 - Navigating All Financial Data Sets and Services

If a GetDataContext() call occurs in the source code for a data set which the user is not subscribed to, a UI sequence is initiated for the user to choose a subscription level for the data set, shown in Figure 16.

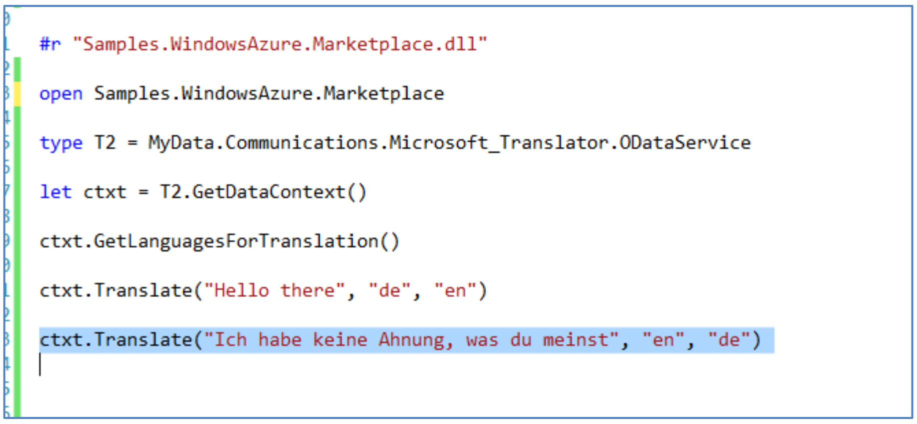


You are not subscribed to the Azure Data Market data set 'Industrial Commodity Statistics Datal
You can subscribe below.
https://datamarket.azure.com/embedded/consent?client_id=fsprovider&response_type=code&

Close
Windows Azure
Marketplace
BROWSE > CHOOSE OFFER > PAYMENT > REVIEW
UNdata
A world of information
Industrial Commodity Statistics Data
United Nations
View Publisher Offer Terms
Unlimited Transactions
Free

Figure 16 - Subscribing to a Data Service

Once a data set or service is chosen, the user can use it through the OData protocol. For example, shows the use of the Microsoft Translator service to translate German to English



```
#r "Samples.WindowsAzure.Marketplace.dll"

open Samples.WindowsAzure.Marketplace

type T2 = MyData.Communications.Microsoft_Translator.ODataService

let ctxt = T2.GetDataContext()

ctxt.GetLanguagesForTranslation()

ctxt.Translate("Hello there", "de", "en")

ctxt.Translate("Ich habe keine Ahnung, was du meinst", "en", "de")
```

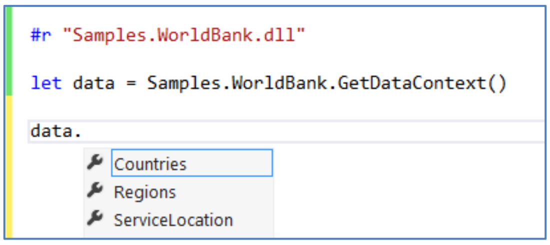
Figure 17 – Code for Using a Data Market Service

4.4 Example: World Bank

One of the major data sets hosted on the Azure Data Market and also available through the Web API api.worldbank.org is the World Bank's aggregation of statistical data sets about countries and regions of the world.

These are time series data covering a vast range of statistical indicators, from finance to health to immigration. The data is organized around the key entities of *country*, *region*, *indicator* and *source*. The individual indicator time series themselves are not small (monthly, quarterly or yearly figures for each indicator for each country or region) but the overall metadata is large – for example, thousands of indicators.

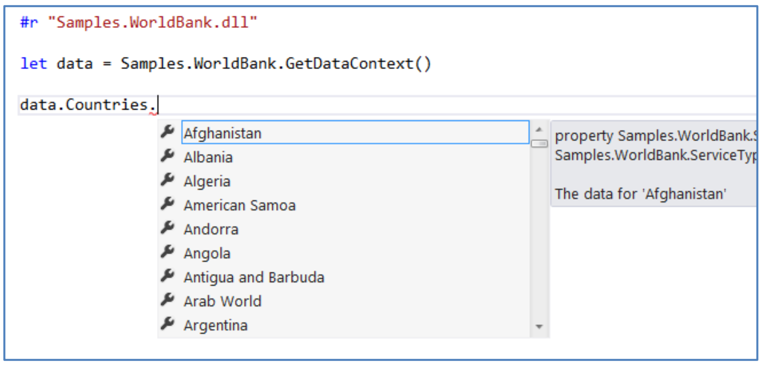
We have implemented a type provider for the World Bank. Figure 18 shows the first exploration of the data space using the type provider in an IDE:



```
#r "Samples.WorldBank.dll"

let data = Samples.WorldBank.GetDataContext()

data.
```

Figure 18 - The World Bank Type Provider

Interestingly, the type provider schematizes the data to the level of individual countries, regions and indicators. For example, Figure 19 shows that each individual country can be "strongly named" and completion lists are available for the full country list. This is immensely useful when scoping in to examine particular countries or regions of interest.



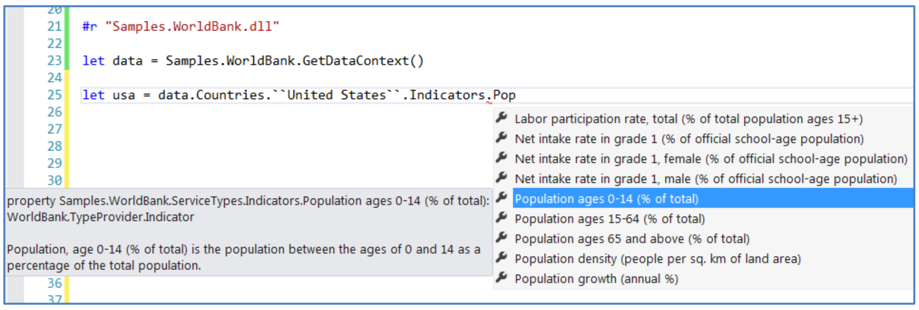
```
#r "Samples.WorldBank.dll"

let data = Samples.WorldBank.GetDataContext()

data.Countries.
```

property Samples.WorldBank.S
Samples.WorldBank.ServiceTyp

The data for 'Afghanistan'

Figure 19 - Countries as Provided Names

Further, the individual indicators can also be strong named and searched. For example, Figure 20 shows the completion list for all indicators for the United States filtered to "Population".



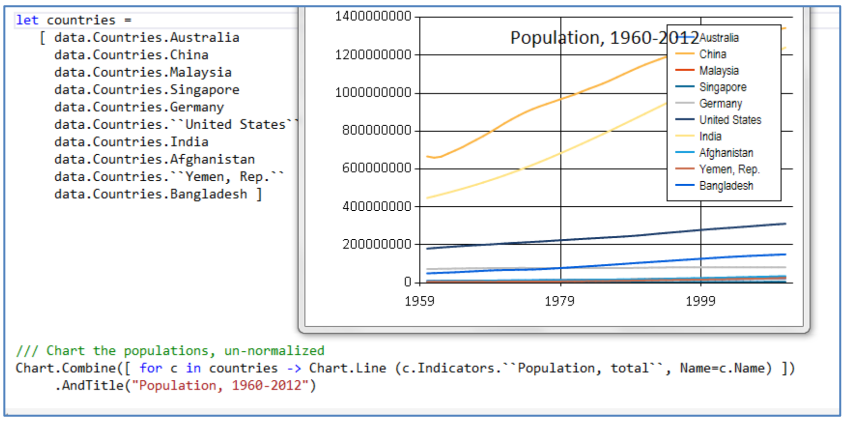
```
#r "Samples.WorldBank.dll"

let data = Samples.WorldBank.GetDataContext()

let usa = data.Countries.``United States``.Indicators.Pop
```

```
Labor participation rate, total (% of total population ages 15+)
Net intake rate in grade 1 (% of official school-age population)
Net intake rate in grade 1, female (% of official school-age population)
Net intake rate in grade 1, male (% of official school-age population)
Population ages 0-14 (% of total)
Population ages 15-64 (% of total)
Population ages 65 and above (% of total)
Population density (people per sq. km of land area)
Population growth (annual %)
```

```
property Samples.WorldBank.ServiceTypes.Indicators.Population ages 0-14 (% of total):
WorldBank.TypeProvider.Indicator

Population, age 0-14 (% of total) is the population between the ages of 0 and 14 as a
percentage of the total population.
```

Figure 20 – Strong naming for Individual Indicators

The combination of F# 3.0, F# Interactive, the World Bank Type provider (with strong naming and assistance for countries and indicators) and the FSharpChart charting library together make for a “super-console” for exploring statistical information about our global world. For example, Figure 21 shows the complete, strongly-typed code used to chart the growth in total population of 10 countries from 1960 to 2012.



```
let countries =
    [ data.Countries.Australia
      data.Countries.China
      data.Countries.Malaysia
      data.Countries.Singapore
      data.Countries.Germany
      data.Countries.``United States``
      data.Countries.India
      data.Countries.Afghanistan
      data.Countries.``Yemen, Rep.``
      data.Countries.Bangladesh ]

/// Chart the populations, un-normalized
Chart.Combine([ for c in countries -> Chart.Line (c.Indicators.``Population, total``, Name=c.Name) ])
     .AndTitle("Population, 1960-2012")
```

Figure 21 - Charting the Population Growth of Ten Countries with the F# 3.0 and the World Bank Type Provider

5 Themes

In Section 4 we have seen four examples of internet-scale information services or protocols integrated into F# via F# type providers. In this section we look at a number of themes in strongly-typed information rich programming, especially when applied to internet-scale information services. These are themes which we have found arise when using the F# type provider mechanism in practice.

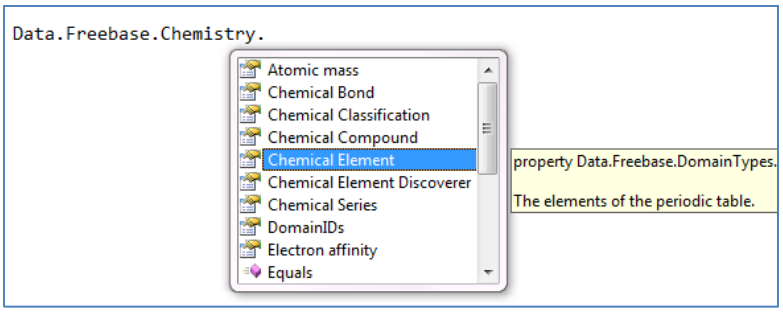
5.1 Theme: Design-time Assistance

During the late 1990s and 2000s, the role of type systems has progressed steadily to encompass not only the traditional goals of reduced error rates (through static checking), memory safety (through soundness of low-level memory access operations) and performance (through elimination of runtime checks) but also code assistance. Modern development environments use types to simplify the implementation of the following:

- Interactive type checking during development ("red squigglies")
- Provision of context-sensitive declaration lists ("auto-completion")
- Type-directed information on gestures such as mouse-hover ("quick info")
- Type and name-directed help systems ("F1 help")
- Name-directed, type-directed or type-safe refactorings

Collectively these are known as the "design-time experience" or "tooling" for typed languages, and the design of any modern programming language must be done with this kind of tooling in mind. Attempts to retrofit a compelling and reliable design-time experience for dynamically typed languages are common but are generally incomplete, unsatisfactory and distort the idiomatic use of the dynamic language.

Some design choices of F# type providers are very much driven with the design-time experience in mind. For example, consider the following completion list shown when using the Freebase type provider mentioned in the previous section – the value of F# type providers lies very much in being able to use design-time tooling to navigate and explore information spaces.



```
Data.Freebase.Chemistry.
```

property Data.Freebase.DomainTypes.

The elements of the periodic table.

Figure 22 - Type providers are designed with tooling in mind

Practicing software designers are very aware that design-time assistance has a strong influence on library design [CA08]. Likewise, design-time tooling has a strong influence on information-space design. Much of the time in developing a type provider is spent in improving the usability of the completion lists in the Human-computer Interaction (HCI) sense of the word, i.e. simple, discoverable and intuitive for a range of expected tasks.

Design-time tooling raises further questions: what other devices and assists could be provided when working with information spaces from programming languages? For example:

- Should the development environment provide **search functionality** for the provided metadata space, e.g. from completion windows? If so, would this affect the provider architecture?
- Should the development environment provide **recommendations** based on the information space?
- Should the development environment display **sample data** for selected elements in the information space? For example, the Freebase provider provides an "Individuals" property for each collection type, which contains named individual entities for easy access through IntelliSense.
- Should the development environment provide **edit** functionality for the metadata itself, on the assumption that the underlying information source allows updates to the metadata?

F# type providers do not implement these specifically. However, individual providers sometimes show sample data in provided documentation, and completion-list filtering provides a simple search mechanism.

Design-time tooling also impacts the technical design of the type provider mechanism. For instance, in addition to checking if a member of a provided type can be resolved we also need a way to list all members of a provided type for IntelliSense completion lists. However, since the graph of provided types can be extremely large (as in the case of Freebase), we want to ensure that these member lists can be calculated lazily as needed by a language service, rather than eagerly. For the same reason, the set of interfaces implemented by a provided type should also be able to be computed on demand.

Finally, this theme raises an interesting question: are type providers a language feature or a tooling feature? Our position is that it is both. From an end-user's perspective, there may not be a difference between the two: where does a language end and a tool begin? For simplicity, we will continue to refer to type providers as a language feature, but this is not meant to downplay the critical role that tooling plays with regard to usability.

5.2 Theme: Schema Change

One of the most important questions when working with rich information spaces is that of *schema change*. From the perspective of the F# type provider mechanism, schema change manifests itself as:

- changes in the *information space schema* and thus the provided component signature, and
- changes in the provided implementations of methods supporting the execution of provided type implementations (i.e. changes in the erasure functions or the provided assemblies)

Before we address this issue, however, we note that there is a strong trend to towards stable, rich information sources delivered through the internet.[2] Further, in typical enterprises there are many information sources with relatively stable schemas that make up the "information base" of the enterprise. This trend is partly based on basic economics: stability attracts developers, developers are crucial for information providers, and so information providers are increasingly in the business of providing schema-stability guarantees in order to attract and satisfy developers.[3] So, while information sources change, in practice a growing number of information sources don't change quite as much as one might think.

The changes mentioned above can happen

- during coding, or
- between coding and execution, or
- during execution.

---

[2] http://www.programmableweb.com/

[3] For example, for-pay data providers on the Azure Data Market (http: //datamarket.azure.com/) will guarantee a period of time during which the schema will not change.

The first and second are only distinct for compiled code, and are conflated for scripting. Type providers address aspects of these changes in three ways:

- When the space of provided types logically changes through the coding process itself, a provider may raise an invalidation signal which resets type-checking for client tools.
- When the space of provided types changes between coding sessions, the strong types and immediate IntelliSense offered by type providers gives good feedback on how to correct the program.
- The (optional) use of type erasure for a type provider can reduce and clarify the set of assumptions baked into provided code, making compiled code more resilient to changes at runtime

We also advocate that type providers come with a schema change specification, i.e. how their behavior is affected by schema change, particularly w.r.t. *source compatibility* and *binary compatibility*. For example, a schema change specification for a database provider may state:

Binary Compatibility: The table and column names of database tables are persisted in compiled code. As a result, if any of the following changes occur, then a runtime error occurs when a column C is accessed at runtime

- the column names changes in the database
- the column is removed from the database
- the column changes its SQL data type

Reordering columns in the database or adding new columns or tables to the database is not a breaking change.

The view that a space of provided types is "just like a library" can be helpful here. We are already familiar with how changing libraries (versioning) exposes the developer to source compatibility and binary compatibility issues both practically and theoretically, and how to factor this into formalisms for software upgrades e.g. [DWE98, ES01, BPN08]. Viewing information sources through the lens of type providers allow us to use the same, common vocabulary when talking about provided spaces of types: we can now meaningfully talk about the source compatibility and binary compatibility properties of a space of provided types under change, just as we would do the same for versions of a library. A good type provider implementation will document these properties clearly, just as a good language, runtime and library will document source and binary compatibility properties. Perhaps one day we may even be able to verify these properties of a type provider implementation.

The F# 3.0 type provider mechanism does not itself provide any means to adjust program execution state based on schema change. This means we normally assume we are working with information sources where there is no schema change during program execution, or in scripting environments where restarting or reloading data is reasonable once schema change occurs. Alternative, extended architectures that adjust program execution state are imaginable, though they would depend greatly on the underlying execution techniques being used. The literature on hot-swapping and dynamic software update is in some ways relevant here e.g. [VBAM09, SHB+07].

5.3 Theme: Connected Programming

Connectivity to the web can be assumed during program development and execution. This means that type providers can typically access the live data sources themselves at design-time, giving the developer an up-to-date schematization during development.

However, in most cases it also makes sense to provide off-line support. For instance, a developer working from home may wish to work with a type provider configured to target an inaccessible corporate database server. Likewise, even the most reliable web services still have occasional downtime. To support a range of realistic scenarios, type providers are frequently designed to locally cache schema information when accessing a live service and to rely on that information if the service is unavailable. Often, the type provider uses a configurable caching policy (e.g. "always connect to the server" vs. "use a cached schema if the service is unavailable") since some developers may prefer not to rely on a potentially stale local cache even when a connection can't be made.

Connected programming also introduces the themes of security and authentication when accessing both schema (at design-time) and data (At runtime). In this report we generally assume that schemas are freely available to all parties at design-time, and that a provider gives a way of specifying credentials for authentication at runtime.

5.4 Theme: Queries

Many information-rich data sources provide special query languages (e.g. SQL for relational databases). Frequently, it is more efficient to use such queries to execute filters, joins, or projections on the server before retrieving data as opposed to executing equivalent logic on the client side. Therefore, it will often be beneficial for type providers to include mechanisms for creating and executing queries over provided types. In some cases, this can be achieved using the standard .NET LINQ IQueryable abstraction [MBB06], but in other cases provider authors may wish to use different abstractions (e.g. if the set of supported query operators differs greatly from those provided by IQueryable sources). Orthogonal to type providers, F# 3.0 also includes a mechanism for embedding arbitrary query languages, which gives type provider authors more flexibility when addressing these concerns.

5.5 Theme: Simplicity and Consistency across Information Spaces

One aim of F# type providers is that that data/information programming experience is consistent. Users can work with disparate sources of data without learning each tool or web API individually. As more type providers are created, we expect that certain patterns and conventions that apply across a wide variety of type providers will emerge. Some design patterns for type providers are provided by Microsoft [SB12].

A preliminary usability study found that users appreciated the ability to access different data sources with different access protocols in a uniform manner, and that they found the IDE integration to be a productivity benefit.

One important aspect to consistency is the way in which different schema elements are mapped into the F# type system. If this is done consistently by different providers, then techniques and code can be reused across information spaces. For example:

- When the information space provides information about units-of-measure, this is usually mapped to the F# type, because units do not add runtime overhead and they provide more precise view of the data.
- When the data source supports inheritance, this can be mapped to .NET class inheritance (provided that it is single-inheritance or that there is a notion of "most important" base class), or to .NET interface inheritance (if there is multiple inheritance).

- If operations of a type provider are I/O intensive, we do not currently map them to asynchronous computations in F# [SPL11]. Avoiding async simplifies explorative programming. Long-running operations must be converted to be asynchronous through the use of a background task or thread. However, if a provider distinguished between long-running operations and operations that can be performed using a local cache, exposing the long-running operations as asynchronous would be a good design choice. Likewise, it is reasonable for a provider to accept a static parameter which optionally allows for the generation of asynchronous calls.

5.6 Theme: Completeness of Providers with respect to Services

The operational characteristics provided by the service should be respected. For example, if the service provides a query language allowing efficient server-side execution of logical operations, the projection of the service into the programming language should also provide a query service.

5.7 Theme: Completeness of the Provider Mechanism with respect to Host Language Constructs

The F# 3.0 type provider mechanism allows for the provision of most, but not all .NET object-model constructs, including classes, interfaces, methods, properties, fields, events and attributes. However, there are some restrictions: for example, F#-specific constructs such as modules, union types and active patterns may not be provided. Also, generic type definitions may not be provided, though instantiations of existing generic type definitions may be, including instantiations with units-of-measure.

One reason for this was simple resourcing: adjusting the compiler for type provision required work and testing for each of these different constructs. Further, to some extent we wished to avoid an eco-system of type providers that relied on F#-specifics, because the type providers themselves may be more generally useful in other contexts. However, over time we expect to lift the remaining restrictions.

Certainly, our experience indicates completeness of possible provided elements w.r.t. the host language (F#) is not a firm requirement for provider mechanisms: one can proceed without it, and in some situations there may be social or interoperability reasons for doing so.

5.8 Theme: Granularity of Schematization

A common theme in embedding data in a strongly-typed way relates to the *granularity* of the schematization of the data. Section 4 showed, for example, a type provider for World Bank data, where schematization is at the granularity of individual countries and indicators, so the question is not just one of granularity of *type* but also of *name*. Further, the question is not just "schematized" v. "unschematized" but rather "how much schema"?

For example, consider the following two ways of accessing the same information, one via the World Bank provider, and the other via the Azure Data Market presentation of the same World Bank data. First, accessing the country and indicator via strong names:

```
#r "Samples.WorldBank.dll"
let data = Samples.WorldBank.GetDataContext()
data.Countries.Australia.Indicators.``Population, total``
```

Next, accessing via a Data Market API that uses strings for country names:

```
#r "Samples.WindowsAzure.DataMarket.dll"
open Samples.WindowsAzure.Marketplace
type T =
   MyData.Business_and_Finance.World_Development_Indicators.ODataService
let ctxtWB = T.GetDataContext()
ctxtWB.GetData("en", "AUS", "SP.POP.TOTL")
```

For the latter, the code is more error-prone, strings such as AUS must be discovered by hand, but the approach is of course correct if these strings are dynamically known values.

Individual providers can, of course, provide multiple granularities over the same data sources – indeed this is very common and supports the transitions between "production" programming over whole sets of data and "investigative" programming against individual items. However, the initial decision of how much schematization to expose is a non-trivial one that requires careful though and design. This is true for all data-space design, but especially true given the new power of scalable type spaces that type providers introduce.

The type provider may also provide different perspective for accessing the data. For example, the World Bank provider makes it possible to access indicators by country (returning a list of year, value pairs). Another useful perspective would be to access indicators by years, in which case the result would be a list of country * value pairs.

5.9 Theme: Providing Additional Metadata (units)

Many schematized data sources include data that is described in terms of physical units of measure (e.g. time in seconds, or mass in kilograms). F# contains unit-of-measure support within its type system [Ken08], so it makes sense to propagate this information in the types provided by a type provider.

Provider authors need to determine whether quantities should be exposed using the raw units from the schematized data source, or whether it is better to convert them to common units (e.g. the SI units contained in the F# standard library). While the latter option may result in better interoperability with data from other sources, conversions can cause a loss of precision. Furthermore, there are occasions where the schema's descriptions contain references to the units used, which might get out of sync with the values if conversion takes place.

6 Some Formal Considerations

In this section, we discuss the properties and criteria that may be of interest when analyzing a programming language formalization of a type provider mechanism.

We will do this in a way that will seem unusual (and disturbingly *informal*) to those of a formal dispensation. This is deliberate: a full formalization of F# type providers is not the focus of this work, and we feel many of the most interesting aspects of type providers (notably, their demonstrated ability to cope with internet-scale information services, and their practical relevance to modern, real-world problems) can easily get lost in a traditional formal treatment of the mechanism. So, in this technical report, we discuss "broader" questions of the assumptions we make as we formalize programming language systems.

6.1 Gamma, The Forgotten

If we consider the basic judgment of typed programming languages:

$$\Gamma \vdash e : \tau$$

In this relation, the traditional research focus has been on e (programs) and τ (representing the types and/or analyses of programs). Relatively little attention is given to Γ, which only takes on interesting structure due to declarations in e and τ. There are some notable and important exceptions, such as work on upgrading software components, dynamic configuration and linking [BPN08]. However because of the general tenancy to focus on expressions and types rather than context, we sometimes jokingly call Γ the "neglected child" or "the forgotten one".

One approach to formalization of F# type providers is to begin to represent the novel aspects of the system in the formal structures. For example, a formal system which captures ones of the key insights of F# type providers (on-demand computation of Γ) is as follows:

```
NamespaceName – N
ClassName – C
PropertyName – P

Γ = VariableContext ProvidedNamespaces       -- environment

VariableContext =
    var : type ... var : type               -- value list

ProvidedNamespaces =
   | N { Classes } …  N { Classes }    -- namespace list

Classes =
   | Delay(Classes)                         -- delayed provided class list
   | C : Class, … ,C: Class                 -- class list

Class =
   | Delay(Class)                           -- delayed provided class description
   | P: Type …   P: Type                    -- class description

type =
   | N.C                                    -- nominal type
   | type → type                            -- function type

expr =
   | new N.C
   | expr.P
   | expr expr
   | \Lambda var. expr
```

With type checking judgments of the form

$$\Gamma \;|\!-\; e : \tau \Rightarrow \Gamma'$$

Type checking both analyzes e and "evaluates" Γ, by reducing the delays present in Γ and its substructures, through name-lookup like the following:

$$\frac{\Gamma\text{'(N)} \;=\; \mathit{Classes}\text{`, else } \Gamma}{\Gamma \;\text{|-}\; \text{new } \mathit{N.C} \;:\; \mathit{N.C} \;\Rightarrow \Gamma\text{'}}$$

$$\frac{\mathit{Clases} \;\text{|-}\; C \Rightarrow \mathit{Classes}\text{'}}{\mathit{Delay}(\mathit{Classes}) \;\text{|-}\; C \Rightarrow \mathit{Classes}\text{'}}$$

$$\frac{\Gamma(\text{N}) \;=\; \mathit{Classes} \qquad \mathit{Classes} \;\text{|-}\; C \Rightarrow \mathit{Classes}\text{'} \qquad \Gamma\text{'(N)} \;=\; \mathit{Classes}\text{`, else } \Gamma}{\Gamma \;\text{|-}\; \mathit{N.C} \;\Rightarrow \Gamma\text{'}}$$

$$\frac{\mathit{Classes} = \{\; \ldots\; C : \mathit{Class} \;\ldots\; \}}{\mathit{Classes} \;\text{|-}\; C \Rightarrow \mathit{Classes}}$$

That is, delays in the substructures of Γ are eliminated as necessary to allow type-checking or name-checking to proceed. Other rules are either standard or straight-forward generalizations of the above.

Then, given Γ |- $e : \tau \Rightarrow \Gamma'$, it is trivial to show that Γ' is the same Γ with some delays removed, and that the number of delay chains removed is at most O(n) where n is the size of e.

For example, consider an environment of logical size 10million (ignoring delays), and an input program of size 10. The number of delays removed from the resulting environment is at most small – if delays are correctly present throughout the initial environment, then only the parts of the environment actually needed by the program are accessed. If each delay-reduction corresponds to a network access to fetch metadata for an information service, then we may have reduced compilation times (and generated code size) by a factor of 100,000 through using delayed environment accesses. Likewise, if each delay represents the generation of some stub code, we may have reduced generated code size by an equally large factor. Computing the environment on-demand can give big wins.

In the formal system above, delays are present throughout the descriptions of namespaces, type definitions (classes) and even individual types. This is also the cases with F# type providers, where every element can be provided on-demand, even to the granularity of metadata such as documentation and definition locations.[4] This is important in practice, where even the added overhead of downloading documentation for each element in a large information space may be prohibitive.

However, the real point is that most formal systems don't need environments of size O(millions) since Γ only arises from declarations in the program itself. That is, adding delayed computations to Γ is not normally useful because program declarations don't include corresponding definitional forms that exploit this power (an exception is the template meta-programming systems discussed in Section 8). As a result, we know of few existing formal systems which use on-demand computation of Γ.

So where did these really big Γ's suddenly come from? Most formal systems make an assumption which is, in certain light, somewhat breathtaking. It is usually written like this:

$$\Gamma_0 = \varnothing$$

That is, most formal systems assume that the initial environment in which programs are checked is empty. Some formal systems like the definition of Standard ML [MTM97] admit a small "initial basis" which is used to interact with the outside world. For many PL researchers, programs exist in a vacuum, and Γ_0 is of literally zero interest.

An empty initial environment is, of course, an immensely useful simplifying assumption for theoretical purposes. However, from a practical point of view (e.g., for those interested in the integration of arbitrary, external information sources), this assumption also represents a somewhat complete denial of the existence of the rich digital world in which modern programs are authored.[5]

The core proposition of F# type providers is that the best strategy we have (at the moment) for taming Γ_0 is based on a two-pronged approach

(a) Rely on the world's information providers to organize Γ_0 into useful networked services, and

[4] There are some small exceptions: for example, the list of (otherwise delayed) types in a namespace is computed eagerly when required for intellisense lists.

[5] Like space, Γ_0 is big. Really big. You just won't believe how vastly, hugely, mind-bogglingly big it is. With apologies to [Ada79]

(b) Simultaneously deploy the full computational power of typed functional languages (and, in particular, ML-like languages) to project those services into the type discipline of those same languages.

Hence the use of a provider model for Γ_0.[6]

6.2 Soundness, Correctness, Contracts and Libraries

It is quickly evident that usual strong notions of "soundness" are weakened by a type provider mechanism. For example:

- A provider may provide component signatures which violate the well-formedness rules of the host language type system. For example, a provider might provider a type hierarchy that contains loops in its inheritance graph, or a component signature that refers to types that do not exist.
- A provider may provide component implementations which do not match their compile-time signatures.
- A provider might provide methods and properties which always fail or go into an infinite loop at runtime
- A provider might rely on external schema information which changes, giving exceptions at runtime when the corresponding provided methods and properties are accessed
- Soundness is also weakened through the use of type-erasure for provided types. Like other .NET languages, F# permits runtime type tests and casts, normally used for interoperability purposes in F# programming. These casts are with respect to erased types, not precise types. Warnings are emitted by the F# compiler in some simple detectable situations where casts are in-exact due to erasure.

Before panic ensues, observe first that *memory safety* remains a strong guarantee. In the case of the .NET implementation of F# 3.0, type providers

[6] We sometimes jokingly refer to F# type providers as the "the cult of Γ_0". That is, F# type providers represent a point of view in PL research that taming Γ_0 and making it useful is where the biggest productivity boosts lie. We seek to tame the messiness and complexities of those oceans of data and metadata in the digital environment, which are both our greatest enemies (when they are not schematized, rapidly changing or not amenable to computational projection) and our trusted companions (when tamed).

describe a component implementations through expression trees of the .NET platform (CIL), which is verified prior to execution in partial-trust scenarios to preserve memory safety and other core properties. Whatever bad things can happen, they aren't *that* bad. Further, the F# 3.0 compiler implements additional checks that verify well-formedness and typing properties of the F# language. However, these well-formedness checks are only performed on-demand by the F# 3.0 compiler.

This means that type soundness is maintained, in the strict technical sense that "the worst that can happen is an exception". Traditional descriptions of type soundness rely on operational semantics which ascribe a "stuck" or "error" state to execution. No new "stuck" states arise, though a provider can be "bad", for example by providing bad component signatures and implementations, and these give rise to compile-time errors. This means that for type providers it seems reasonable to talk about *conditional* type soundness – that is, a program is sound up to the behavior of the provider, and in particular the soundness of the component signatures and implementations that are provided. The IL-level type soundness of generated code becomes conditional on the soundness of the erasure functions and assembly implementations provided by individual type providers used in a compilation. Assuming the provided expressions are valid, the generated code is sound. The F# compiler checks some type correctness conditions of the provided expressions, and additional checks are provided by the helper code used to implement individual type providers.

However, soundness (or, if you like, correctness) of language-integrated information sources clearly goes beyond mere memory and type safety. Type providers are effectively providing on-demand implementations of library implementations (often based on external schemas), so soundness for a type provider is closely related to soundness for a traditional library, a topic which is largely unexplored in the academic literature (there are endless papers on the soundness of languages, but few papers on the soundness of libraries). For the specific application examples shown in this paper, some of the notions of soundness we're interested are:

- For all providers using remote data sources, no "deserialization" exceptions occur while the schema remains unchanged; and no "wrong type of data" exceptions occur when writing in-memory objects back to external data sources through provided types.

- For a database provider, no "database property not found" exception occurs unless the table column is removed.
- For the World Bank provider, no "country not found" exception occurs (even if a geopolitical schema change occurs, old country codes remain valid).
- For the Azure Data Market provider, no "service not found" exception occurs, unless the Data Market providers withdraw a service.
- For the Freebase provider, no "chemical elements don't have an atomic number" exception occurs.
- For the Azure Data Market provider, no "service expects 3 parameters not 2" exception ever occurs (because the service schema is guaranteed not to change shape).
- For the R provider (section 7), no "R package not found" exception occurs if the R package is installed on the machine at runtime

That is, we do not expect "WebData.Chemistry.ChemicalElements" to fail at runtime with a "ChemicalElements not found" exception. However, the provider may specify that it is possible to fail with a "No Web Connection Exception". This kind of correctness property is again conditional on individual type providers and is thus hard to characterize in a general way – it depends on the specification of each provider.

This means the behavior of the provided types, properties and methods must be specified with respect to schema change. In particular we recommend that provider writers specify the binary-compatibility and source-compatibility properties induced by schema change in the external information store, as discussed in previous sections. We have discussed this in Section 5.

7 Further Applications

In the course of our experiments with type providers, we have mainly explored information spaces which fit into a few broad categories:

1. **Remote data sources** such as databases and web-based services.
2. **Structured file formats**: such as Excel, CSV, TSV or netCDF.

3. **DSL texts**, such as regular expressions (where named groups provide the structure) or printf-style format strings (where placeholders provide the structure).
4. **Code providers** such as providers for interoperating with R or Python.

In the remainder of this section we will highlight some interesting aspects of a few particular examples of type providers that we have built.

7.1 SQL + LINQ + Type Providers

Databases are among the most commonly accessed data sources in the programming world. The release of .NET 3.5 included LINQ-to-SQL for accessing SQL Server databases from .NET code, which combined a code-generator and runtime libraries for data access as well as complementary extensions to VB.NET and C# to make in-language query writing more fluent. Since then, other persistence technologies (such as the Entity Framework) have been integrated with the .NET platform using a similar approach.

To streamline the process of using these technologies from F# we have built a family of type providers and a query implementation. These generative type providers run the respective code generators, producing .NET assemblies containing types that are embedded into the assembly that use them. The query implementation is part of F# 3.0's standard library, and provided a query DSL for querying any .NET IEnumerable or IQueryable data source. The query DSL includes both the standard projection and filtering operations (as exposed by C# and VB.NET's LINQ implementations) and other built-in IQueryable features such as aggregation.

By using these type providers, F# developers can write code that manipulates strongly typed database entities from a single F# script, without ever having to explicitly generate code or interrupt their normal workflows.

This family of providers touches on many of the themes from Section 5:

- The type providers optionally cache schema data locally in case the user wants to code against the type provider when there's not a live connection to the database.
- A query DSL exposes database queries in language-integrated fashion while ensuring that queries are performed on the database when possible.
- The family of providers uses a consistent set of naming conventions and data representations, giving users a familiar experience as they move from

one provider to another. The same conventions are also used in our type providers for accessing web services, where applicable.

- Authentication can be performed using credentials stored in configuration files or with integrated authentication
- The providers cover the complete set of ad-hoc options used by the wrapped code generators, such as name pluralization options.
- The user can choose between type providers that connect directly to the database (for DB-first development), or that use separate metadata files (.dbml or .edmx files) to convey the schema.
- When using separate metadata files, the providers react to schema changes by invalidating stale type information.

7.2 Example: Windows Management Interface

In this section we discuss the use of F# type providers to give strongly-typed language integration for the "management" information made available by a modern operating system. In this case we look at the Windows Management Instrumentation (WMI) information provided by Windows machines, a superset of the industry-standard CMI machine management information.

WMI provides an operating system interface through which instrumented components provide information and notification. WMI also supports a query language (WQL), a subset of the standard ANSI SQL with minor semantic changes. WMI is usually accessed via PowerShell or via the System.Management .NET API. WMI is often accessed from scripting languages such as PowerShell, since it is hard to give a strongly typed and navigable API to WMI information from a typed language. Thus, even when using C# to program against WMI, the benefits of strong typing are lost. Figure 23 shows such an example. Not only is the code longer than necessary, the programmer also loses the benefit of IDE autocompletion. Furthermore, the programmer must be well-versed in WQL in order to even get started. In contrast, writing a WMI type provider for F# is relatively simple, and using it is simpler still. Note the simplicity and strong typing in the F# code in Figure 24 as compared to the C# code in Figure 23.

```
using System;
using System.Management;
public class Connect
{
    public static void Main()
    {
        var scope = new ManagementScope("\\\\localhost\\root\\cimv2");
        scope.Connect();

        //Query system for Operating System information
        var query = new ObjectQuery("SELECT * FROM Win32_OperatingSystem");
        var searcher = new ManagementObjectSearcher(scope,query);

        var queryCollection = searcher.Get();
        foreach ( ManagementObject m in queryCollection)
        {
            // Display the computer information
            Console.WriteLine("Computer Name: {0}", m["csname"]);
            Console.WriteLine("Windows Directory: {0}",
                             m["WindowsDirectory"]);
            Console.WriteLine("Operating System: {0}",  m["Caption"]);
            Console.WriteLine("Version: {0}", m["Version"]);
            Console.WriteLine("Manufacturer: {0}", m["Manufacturer"]);
        }
    }
}
```

Figure 23. Using the Windows Management Interface directly in C# using a SQL-like query language with embedded strings and without type information.

```
open System
open System.Management
open Microsoft.Management.TypeProvider // this is the WMI provider

// Display the computer information
for m in LocalMachine.Win32_OperatingSystem do
    printfn "Computer Name: %s" m.CSName
    printfn "Windows Directory: %s" m.WindowsDirectory
    printfn "Operating System: %s"  m.Caption
    printfn "Version: %s" m.Version
    printfn "Manufacturer: %s" m.Manufacturer
```

Figure 24. The same operations as in Figure 23, but using the WMI type provider

7.3 SharePoint: A Language-Integrated Content Management System

SharePoint is a popular platform for content and document management and corporate intranet development [Mic12a]. SharePoint 2010 includes a new "Client Object Model" which makes it possible to programmatically access SharePoint sites from managed code written in .NET languages. In particular, the APIs provide access to a SharePoint site's structured lists, which contain most of the interesting content for the site. However, because the structure of those lists is dynamically configured from within the SharePoint site itself, the APIs require the programmer to select lists and fields using names stored in strings and to unbox field values from objects to the appropriate types. For more advanced use cases, list entries can accessed by writing queries in SharePoint's CAML query language[Mic10], but this typically requires creating the query by hand. Furthermore, CAML's semantics are also somewhat unique, making the standard .NET IQueryable abstraction a poor fit.

We have written an F# type provider for accessing SharePoint sites which wraps the standard .NET client object model libraries and alleviates most of these issues. This provider exposes each SharePoint list as a provided type, with a statically typed provided property wrapping each of the list's fields. We have also created a SharePoint query DSL to make it easy to write and execute CAML queries over those lists in a statically typed, safe manner:

```
// Connect to the F# user group SharePoint site
type FSUG = SharePoint.TypeProvider.SharePointSite<"http://fsug.org", usernam
let context = new FSUG.DataContext()

// Create a CAML query to get interesting meeting details
let meetings =
    sharepoint {
        for mtg in context.Site.RootWeb.``FSUG Meetings``.ItemCollection do
        where (mtg.Description.Contains "F#")
        expand mtg.ID
        expand mtg.Speaker
        expand mtg.Description
        expand mtg.Category
        orderByDesc mtg.ID
    }
```

Figure 25 - Accessing SharePoint using a type provider

In addition to those domain-specific aspects, the SharePoint type provider also deals with common concerns:

- The provider supports integrated authentication and password-based authentication.
- The provider supports lazy loading (that is, delaying the retrieval of unused fields from lists).

7.4 CsvFileProvider: Strongly Typed Tabluar Data

Tabular data formats are common in many scientific and financial programming contexts. Accessing the data in these files is often made cumbersome by the inability of the programming language to take advantage of schema information that is present in the headers of these files (e.g. column names), forcing programmers to either create and populate one-off data types on a per-file basis or to access the data in columns ordinally.

We have created a type provider for reading tabular data from delimited files (typically comma- or tab-separated files). This provider includes features such as producing unit-of-measure-annotated data, which provides important safety benefits for scientific and financial programmers.

One interesting question that arises when building such a type provider is how the schema for a file should be obtained (since that information is not always present in the file itself). There are several alternatives, each of which may be most appropriate in some contexts:

1. Give the option for the type information to be added to the header row (e.g. following the column name). This is an invasive requirement, but

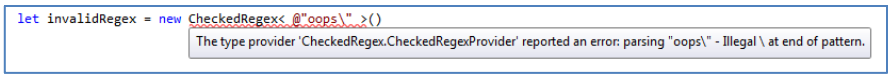
forces the user to be explicit and keeps the metadata close to the data that it describes.

2. Give the option to attempt to infer the types from the data (c.f. the Rows to Scan option for the ODBC Excel reader). Using this approach, users do not need to make changes to their files but it's possible that the data that is currently available is not wholly representative (e.g. an optional column may happen to have values for each row, or a column of text data may happen to contain only values that could be interpreted as integers) and therefore will be inferred incorrectly.
3. Give the option for the user of the type provider to specify the type information in the static arguments to the type provider. This does not place any additional burden on the creator of the data, but does require some effort from the provider's user. Furthermore, since the type information is external to the file itself, it is perhaps more likely that it will become out of sync with the file's data (e.g. a schema change may go unnoticed by the type provider user), though the type provider can statically check for some consistency properties.

7.5 RegexProvider: Strongly Typed Language-Integrated Regular Expressions

As an example of how type providers scale down to even simple structured data, we created a type provider for regular expressions. This provider takes a regular expression as a static argument (using the standard .NET regular expression format), and provides two benefits:

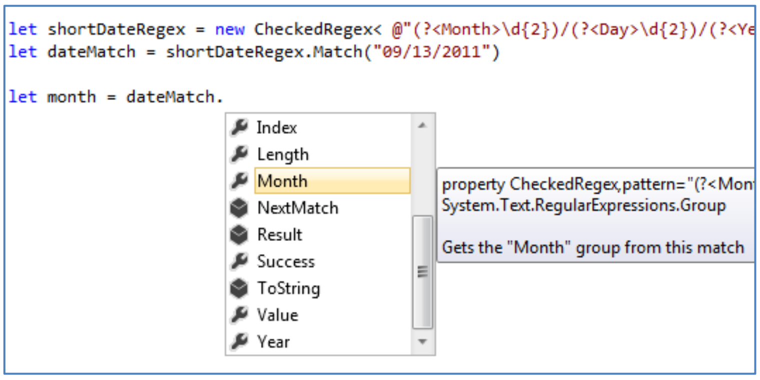
1. Syntactically invalid regular expressions produce errors at compile time, not runtime.



```
let invalidRegex = new CheckedRegex< @"oops\" >()
```

The type provider 'CheckedRegex.CheckedRegexProvider' reported an error: parsing "oops\" - Illegal \ at end of pattern.

2. Named groups in matches are accessible as properties of the match object, allowing users to easily access them through IntelliSense and alleviating users from having to use string literals which are only checked at runtime.



```
let shortDateRegex = new CheckedRegex< @"(?<Month>\d{2})/(?<Day>\d{2})/(?<Ye
let dateMatch = shortDateRegex.Match("09/13/2011")

let month = dateMatch.
```

property CheckedRegex,pattern="(?<Mon
System.Text.RegularExpressions.Group

Gets the "Month" group from this match

The provider itself is quite compact, totaling approximately 120 lines of code, excluding a shared library of utility functions used by most of our type providers.

7.6 Interop providers

In the course of our work on F# type providers we have implemented experimental providers that are intended to ease interop with other programming languages and environments:

1. **COM provider:** this provider allows COM components to be used directly from F# without needing to go through an explicit conversion to CLR types.
2. **R provider:** this provider allows R dataframes to be used directly from F#. A more general R provider for F# is also now being developed independently [Man12].
3. **C provider (experimental!):** this provider allows inline C code to be included in an F# program, which is then compiled at design-time and can be executed at runtime. In principle, this should make it possible to utilize specific CPU instructions that aren't supported by the CLR (such as certain SSE extensions, for example).

A natural question then arises whether F#'s CLI/C#/.NET interoperability could itself be ultimately built by using a type provider. In the limit, this would be possible. However, F# has numerous typing rules that relate specifically to CLI constructs and these would also need to be considered.

8 Summary and Related Work

If the web and multi-core were the pervasive issues of programming in the first decade 21st century, the biggest challenge for the programming landscape of the next 10 years is to integrate internet-scale information services directly into programming languages in ways that improve programmer productivity, performance, application robustness and application maintainability. The work we have described here represents a contribution (though by no means a final one) in that direction.

8.1 Summary

In this report, we have described the F# 3.0 type provider mechanism, illustrating the following key points:

- The mechanism scales to information sources containing extremely large quantities of metadata.
- The mechanism can be applied effectively to internet-scale information services including web data protocols (OData), web ontologies (Freebase), web-based data markets (Azure Data Market) and massive information services (World Bank).
- The mechanism enables the use of code completion and interactive type checking to increase programmer efficiency when working with rich information sources.
- The mechanism interacts with strongly typed tooling such as documentation assistance and completion lists and uses these as the primary way of exploring and understanding the information sources being used.
- The programming experience is code focused. Programmers need not interrupt their coding tasks and switch to a design tool or a code generator, or manually check whether the data schema has changed. Because of this, type providers integrate well with strongly typed scripting such as scripting with F# Interactive.
- The programming infrastructure is neutral with regard to the protocol and data formats used. F# itself, as a language or toolset, has no specific knowledge of OData, Excel, or Freebase.
- The mechanism uses an open architecture, so one can easily add new type providers that consume a different kind of schematized data.

- The mechanism can integrate with advanced features of typed programming languages such as units of measure. They multiply the value of these features because of the sheer quantity and importance of unitized data available in external information sources.

8.2 Why a Strongly-Typed, Functional-First Language?

This report shows clearly the breadth and usefulnesss of the F# 3.0 type provider mechanism. We now address a final question: why F#? Or, more generally, why are type providers such a good fit for strongly-typed, functional-first programming?

We have identified several reasons for this

- **Type inference.** F# and all strongly-typed functional languages use type-inference extensively. Type providers fit extremely well with the highly type-inferred languages. In none of the code samples shown have we needed to write type names explicitly: they have always been inferred from the data. F# type inference requires considerably fewer type annotations than C#, C++ or Scala, and many fewer than Java. (Note, for example, that type annotations are not needed on return types for F# functions)
- **The Best of OO, the Best of Functional.** Type providers generally use a mix of functional and OO mechanisms to represent provided information spaces
    - Individual provided entities are represented as object-oriented types, making extensive use of "dot" property notation. OO techniques are unrivalled for representing large library designs, so it makes sense that they are used for this role in F#.
    - Provided collections use "functional programming" types such as sequences.
    - Provided queryable collections use the LINQ "functional query" paradigm of .NET [MBB06].
    - Provided services sometimes make use of the .NET Task<T> [LSB09] or F# Async<T> programming support [SPL11], both of which interact well with compositional functional programming techniques.

- **Interactive Execution.** The combination of type providers with a REPL (in our case F# Interactive) gives a code-focused, data-scripting experience.
- **Prior data paucity.** Functional programming languages have historically been data-deficient, in the sense that accessing external data sources has been hard and has broken the strong-typing paradigm of these languages. The needy feel the benefit of type providers very strongly!
- **Visual Tooling.** The examples in this paper demonstrate the importance of auto-completion as a way of navigating data spaces. F# 2.0 implemented quality "base-level" visual tooling for F#, including auto-complete and some other coding assists. F# 3.0 builds on this to use these mechanisms in conjunction with type providers. This indicates the growing importance of coding assists in language/tool design, as discussed in Section 7.
- **Meta-programming** foundations. At the API level, F# type providers utilize several .NET and F# meta-programming idioms, including .NET System.Type meta-programming and F# quotations [Sym06]. The pre-existence of these facilities was an important factor in making the type provider implementation possible within reasonable resource constraints.

8.3 Related Work

The topic of data access and the integration of data with programming languages has given rise to a vast literature and a vast body of applied work. In this section we review some of the work more closely related to integrating data into strongly-typed programming languages.

8.3.1 Static Libraries and Code Generation

Section 2 mentioned three techniques that are traditionally used to bridge programming languages to information sources:

(a) hand-written static libraries, and

(b) generated static libraries, and

(c) dynamically-typed information representation.

Most related work uses one of these approaches. First, some information sources have relatively small, stable schemas and can be accessed through a bespoke library. Twitter is a good example.

Next, the industry standard technique for schematized data is to use a code generator, often based on input from a "designer". For example, LINQ [MBB06] allows programmers to write SQL queries directly within a typed framework. Programmers must use either a designer tool and/or a code generator to map from the database schema to .NET types. Using a designer tool is not an optimal experience for developers, who must learn to effectively use yet another tool, context switch between the design tool and the code, and keep the database schema in sync with the generated code. The second option, code generation, is also problematic: it adds yet another step to the build process, which means that automated builds must have the tool or its artifacts, it interacts badly with source control, particularly when the generated code is large and should be cached and shared among a development team, and it leads to either code bloat or the need for (yet another!) tool for pruning the compiled artifacts.[7]

8.3.2 Compile-time meta-programming

F# type providers are a novel form of compile-time meta-programming. Other approaches to compile-time meta-programming are normally based on code generation or macros, including CamlP4 and Template Haskell. These systems are not generally used for information-integration – to quote [DP10]

"Static metaprogramming" is compile-time code analysis and synthesis. It has many applications, such as (from simple to complex): defining abbreviations, generating boilerplate from type definitions, extending the language syntax, and embedding DSLs.

Further, these systems all require eager download of all metadata in order to perform code generation. This means they can only be applied to a subset of the internet-scale information services described in this report, at least without further modification. In addition, these systems seem almost "too powerful" for the specific tasks of information-integration, admitting a wide range of extra macro-like capabilities into the normal use of the language. In

[7] Note that this last point is not a hypothetical: the authors have been informed of cases where the generated code for an enterprise system is so large that it cannot fit within the limits of a .NET process!

contrast, F# type providers have been specifically limited to focus on information service integration.

8.3.3 Open Type Systems

One language system that appears close in spirit to F# type providers is the JVM-based language Gosu and its "open" type system [McK12], which is in some ways similar in flavor to type providers. The system has been applied to the language-integration of some enterprise data standards such as XML. To our knowledge, Gosu has not yet been applied to the broad range of scalable information services described in this report nor integrated with the type-inference capabilities of a strongly-typed functional language. The Jolie orchestration language [MGLZ07] also has adaptors for types from XML.

8.3.4 Query Programming

Language integration for internet-scale data sources relies heavily on language-integrated query programming, especially LINQ [MBB06]. Recent applied programming languages research in the area has looked at "avalanche safety" [GRS10]. This can also be applied to LINQ [SBG+10] and would thus also be applicable to the query implementations in this report.

Query integration into strongly-typed languages remains a challenging area with many under-explored corners: for example, LINQ implementations are still difficult to write, and do not cover many important query paradigms such as OLAP in a strongly-typed way [NNT01]. Query programming based on homogeneous meta-programming remains fragile and prone to runtime failures when functions available on the client are not available on the server, or are given a different semantic interpretation.

8.3.5 Pluggable Type Systems

Many researchers have investigated the idea of pluggable type systems [PAC+08] and a variety of lightweight mechanisms in which new type systems could be layered on top of existing ones. There is some similarity here: type providers are "plug-ins" to the library-import logic of a language (loosely speaking, the representation and rules covering Γ_0 in a language formalization), and pluggable type systems are "plug-ins" to the typing rules of a language (loosely speaking, the representations and rules for τ in a formalized type system). In this sense, the approach taken by JavaCOP [MME+10] is analogous to ours, in that it allows plugin writers to give a declarative specification of an analysis, which is then utilized by the JavaCOP's main analysis framework. However, this approach has not been applied to the language integration of internet-scale information services.

8.3.6 Highly Generic Type Systems

One way to approach information representation problems is by increasing the power of type-level computations that can be performed allowing the execution of full programs during compilation. For example, Ur [Chl10] includes a powerful system of type-level computations over record types that can be used to represent the schema transformations that occur in some database and web programming.

These mechanisms are powerful extensions to type systems and highly suited to playing a role in information-rich programming. Because they extend the type system, they come with all the usual tradeoffs exhibited by such additions (e.g. increased complexity v. increased expressivity). These mechanisms would combine nicely with a provider model of the kind we have described here. Further, a provider model of some kind is clearly necessary in order to apply these techniques to the internet-scale information services we have examined in this report.

8.3.7 Dependent Types

F# type providers are a manifestation of a "pseudo"-dependent type system, where type schemas are dependent on information fully available at compile-time. This is, however, only a passing similarity: the types provided by an F# type provider do not depend on values computed at runtime, and pure dependent type systems can't base type-level computations on external information such as schemas.

8.3.8 Staged Computation

F# type providers are a novel form of staged computation [TS97], where one phase of computation can explicitly generate elements of the typing environment for the next phase using a compositional plug-in model. In one sense, this is exactly what is happening with F# type providers: the environment is being computed by executing programs in the compilation stage. However, the application areas we have explored in internet-scale information sources are very, very different to traditional uses of staged computation, and it is not clear that any of the staged computation mechanisms so far designed or implemented could work effectively for these kinds of application areas. Further the F# type provider mechanism is deliberately weaker than full staged computation – it is for useful metadata representation at compile-time, and adds no complexity to F# runtime execution: it is a purely static mechanism. In contrast, traditional uses of staged computation are for efficient code-generation at runtime.

Acknowledgements. Over the past 3 years we have discussed F# type providers with just about everyone we could find: data experts, information service providers, analytical programmers, statisticians, programming language designers, PL researchers, tool developers, data scientists, family members, children, people in coffee shops. Almost every one of these conversations has led to interesting insights into what it would mean to "embed the world of digital information" into a programming language. To name and thank all these people would not be possible here. However, we particularly thank Ralf Herbrich for his early help and advice on the design of the F# 3.0 type provider mechanism and the fascinating and stimulating conversations that resulted.

References

[1?] "DBLP:conf/ecoop/2008" not found in database

[Ada79] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979.

[BEP$^+$08] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1247–1250, New York, NY, USA, 2008. ACM.

[BPN08] Gavin M. Bierman, Matthew J. Parkinson, and James Noble. Upgradej: Incremental typechecking for class upgrades. In *ECOOP*, pages 235–259, 2008.

[CA08] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries*. Microsoft .NET Development Series. Addison-Wesley, 2008.

[Chl10] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. *SIGPLAN Not.*, 45(6):122–133, June 2010.

[COP12] Michael J. Carey, Nicola Onose, and Michalis Petropoulos. Data services. *Commun. ACM*, 55(6):86–97, June 2012.

[DP10] Jake Donham and Nicolas Pouillard. Camlp4 and Template Haskell. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 6:1–6:1, New York, NY, USA, 2010. ACM.

[DWE98] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 341–361, New York, NY, USA, 1998. ACM.

[ES01] S. Eisenbach and C. Sadler. Changing Java Programs. In *IEEE Conference in Software Maintenance (ICSM 2001)*, Florence, Italy, November 2001.

[fsh12] F# 3.0 Sample Pack, March 2012. http://fsharp3sample.codeplex.com/, retrieved 1 August 2012.

[GRS10] Torsten Grust, Jan Rittinger, and Tom Schreiber. Avalanche-safe linq compilation. *Proc. VLDB Endow.*, 3(1-2):162–172, September 2010.

[Ken08]Andrew Kennedy. Types for units-of-measure in F#: invited talk. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, ML '08, pages 1–2, New York, NY, USA, 2008. ACM.

[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[Man12] Howard Mansell. An F# type provider for R, 2012. https://github.com/BlueMountainCapital/FSharpRProvider, retrieved 1 August 2012.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Int. ACM Conf. on Mgmt. of Data*. ACM, 2006.

[McK12] S. McKinney. Gosu's secret sauce: The open type system, November 2012. http://guidewiredevelopment.wordpress.com/2010/11/18/gosus-secret-sauce-the-open-type-system, retrieved 1 August 2012.

[MGLZ07] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a Java orchestration language interpreter engine. *Electronic Notes in Theoretical Computer Science*, 181(0):19 – 33, 2007.

[Mic10] Microsoft. SharePoint Query Schema, May 2010. http://msdn.microsoft.com/en-us/library/ms467521, retrieved 1 August 2012.

[Mic12a] Microsoft. Sharepoint - collaboration software for the enterprise, 2012. http://sharepoint.microsoft.com/, retrieved 1 August 2012.

[Mic12b] Microsoft. Windows Azure Marketplace: A one-stop shop for premium data and applications, July 2012. https://datamarket.azure.com/, retrieved 1 August 2012.

[MME$^+$10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Trans. Program. Lang. Syst.*, 32(2):4:1–4:37, February 2010.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[NNT01] Tapio Niemi, Jyrki Nummenmaa, and Peter Thanisch. Constructing olap cubes based on queries. In *Proceedings of the 4th ACM international workshop on Data warehousing and OLAP*, DOLAP '01, pages 9–15, New York, NY, USA, 2001. ACM.

[PAC$^+$08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[SB12] Don Syme and Keith Battocchi. Tutorial: Creating a type provider in F#, January 2012. http://msdn.microsoft.com/en-us/library/hh361034%28v=vs.110%29.aspx, retrieved 1 August 2012.

[SBG+10] Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. Thirteen new players in the team: a ferry-based linq to sql provider. *Proc. VLDB Endow.*, 3(1-2):1549–1552, September 2010.

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[SHB+07] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), August 2007.

[SPL11] Don Syme, Tomas Petricek, and Dmitry Lomov. The F# asynchronous programming model. In *Proceedings of the 13th international conference on Practical Aspects of Declarative Languages*, PADL'11, pages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.

[Sym06] Don Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, ML '06, pages 43–54, New York, NY, USA, 2006. ACM.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, pages 203–217, New York, NY, USA, 1997. ACM.

[VBAM09] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. Advanced runtime adaptation for Java. *SIGPLAN Not.*, 45(2):85–94, October 2009.

Appendix A. The Low-Level Type Provider API

An approximate C# view of the type provider interface is shown in Figure 26. A type provider implementation must define a subclass conforming to this interface and a metadata attribute (on the assembly and the subclass) to denote its status as a type provider.

Appendix B. Implementing a Type Provider

Type provider implementations are typically 100–1000 lines of code.

Figure 27 shows a simple example of a provider that provides 100 types which each expose a few constructors, properties, and methods. Note that this code depends on a library which simplifies some aspects of writing a type provider (see the F# 3.0 Sample Pack [fsh12]).

```
using System;
using System.Reflection;

interface IProvidedNamespace
{
  /// Get the namespace name the provider injects types into.
  string NamespaceName { get; set; }

  /// The sub-namespaces in this namespace.
  IProvidedNamespace[] GetNestedNamespaces();

  /// Get the types in the namespace.
  Type[] GetTypes();

  /// Resolve a type name in the namespace.
  Type ResolveTypeName(string typeName);
}

interface ITypeProvider
{
  /// Get the namespaces provided by this type provider.
  IProvidedNamespace[] GetNamespaces();

  /// Get the static parameters for a provided type.
  ParameterInfo[] GetStaticParameters(Type typeWithoutArguments);

  /// Apply static arguments to a provided type.
  Type ApplyStaticArguments(Type typeWithoutArguments,
                            string[] typePathWithArguments,
                            object[] staticArguments);

  /// Get the implementation of a call to a provided method.
  Expr GetInvokerExpression(MethodBase syntheticMethodBase,
                            Expr[] parameters);

  /// A type provider may raise this event when an assumption
  /// changes that invalidates the resolutions so far reported
  /// by the provider
  event System.EventHandler Invalidate;

  /// Get the physical contents of the generated provided
  /// assembly.
  byte[] GetGeneratedAssemblyContents(Assembly assembly);

}
```

Figure 26. The type provider interface

```
open System
open System.Reflection
open Samples.FSharp.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()

  // Make one provided type, called TypeN
  let makeOneProvidedType (n:int) =

      // This is the provided type. It is an erased provided type, and
      // in compiled code will appear as type 'obj'.
      let t = ProvidedTypeDefinition(thisAssembly,namespaceName,
                                     "Type" + string n,
                                     baseType = Some typeof<obj>)

      // Add documentation to the provided type.
      t.AddXmlDocDelayed (fun () -> sprintf "This provided type %d" n)

      // This is a provided static property. A get of this property will
      // always evaluate to the string "Hello!".
      let staticProp =
          ProvidedProperty(propertyName = "StaticProperty",
                           propertyType = typeof<string>,
                           IsStatic=true,
                           GetterCode=(fun args -> <@@"Hello!"@@>))

      // Add documentation to the provided static property.
      staticProp.AddXmlDocDelayed(fun () -> "This is a static property")

      // Add the static property to the type.
      t.AddMember staticProp

      // This is a provided constructor with no parameters.
      let ctor =
          ProvidedConstructor
              (parameters = [ ],
               InvokeCode= (fun args -> <@@ "The obj data" :> obj @@>))

      // Add documentation to the provided constructor.
      ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

      // Add the provided constructor to the provided type.
      t.AddMember ctor

      // This is a provided constructor with one parameter.
      let ctor2 =
          ProvidedConstructor
            (parameters = [ ProvidedParameter("data",typeof<string>) ],
             InvokeCode= (fun args -> <@@ (%%(args.[0]) : string) :> obj @@>))
      ctor2.AddXmlDocDelayed(fun () -> "This is a constructor")

      // Add the constructor to the type.
      t.AddMember ctor2

      // This is an instance property. Getting this property will get
      // the length of the string which is the representation object.
      let instanceProp =
        ProvidedProperty(propertyName = "InstanceProperty",
                         propertyType = typeof<int>,
                         GetterCode= (fun args ->
                           <@@ ((%%(args.[0]) : obj) :?> string).Length @@>))

      instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")

      // Add the instance property to the type.
      t.AddMember instanceProp

      // This is an instance method with one parameter. This method will
      // get the character in the representation at the given index.
      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

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

 // And add them to the namespace
  do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()
```

Figure 27. Demonstration type provider