Themes in Information-Rich Functional Programming for Internet-Scale Data Sources

Don Syme, Keith Battocchi, Kenji Takeda
Microsoft Research
Cambridge, UK
dsyme@microsoft.com

Donna Malayeri
Microsoft Corporation
Redmond, WA, USA
donnam@microsoft.com

Tomas Petricek
University of Cambridge
Cambridge, UK
tomas.petricek@cl.cam.ac.uk

Abstract

The F# language includes a feature called "F# 3.0 Type Providers" to support the integration of internet-scale information sources into a strongly typed functional-first programming environment. In this position paper we describe the key themes in information-rich functional programming that we have observed during this work. Our contribution is to document these themes and highlight future challenges and opportunities, in the context of a recently released, practical, open-source system for information-rich functional programming. We believe that this area is rich in excellent opportunities for future language and tooling research, information-space integration and schematization techniques.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – data structures, D.2.12. Interoperability – data mapping.

General Terms Languages, Design

Keywords functional programming; Freebase; semantic web; data services; connected programming; LINQ; ontology; F#

1. Introduction

Over the last 3 years the F# language team have designed and implemented a programming language feature called "F# 3.0 Type Providers" to support the integration of internet-scale information sources into a strongly typed programming environment [8]. In this position paper we describe the key themes in information-rich functional programming that we have observed during this work. Our contribution is to document these themes and highlight future challenges and opportunities, in the context of a recently released, practical, open-source system for information-rich functional programming, We believe that this area is rich in excellent opportunities for future language and tooling research, information-space modeling, schematization techniques, and usability analysis.

By way of introduction, 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: a representation of a provided component signature that acts as the programming interface to that information space, and a provided component implementation of the component signature. Both the signature and implementation are computed on-demand (i.e. lazily), when required by the language tooling for the program being compiled. The implementation is given by a pair of erasure functions giving representation types and representation expressions for the provided types and values respectively. Put most simply, a type provider is an adapter component that reads schematized data and services and transforms them into types in the target programming language in an on-demand and scalable way. Type providers are about using an on-demand provider model for the "type import" logic of the host language compiler or tooling.

A mini-formalization of a calculus related to type providers is described in [8], along with the low-level API for type providers.

The specific F# 3.0 implementation of the type provider design has the following characteristics:

- The mechanism scales to information sources containing extremely large quantities of metadata.
- The examples in [8] show how 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 web 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 and integrates well with strongly typed data scripting REPL environments (e.g. F# Interactive).
- The programming infrastructure is neutral with regard to the protocol and data formats used. F# itself, as a language, has no specific dependency on external data sources or formats.
- The mechanism uses an open architecture, so one can easily add new type providers that consume a different kind of schematized data.
- The mechanism integrates technically with advanced features of type systems such as units of measure. Further, it multiplies the value of these features because of the quantity and importance of unitized data available in external information sources.

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

An example of using a type provider to explore the "Sports" section of the Freebase knowledge graph is shown in Figure 1.

1.1 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-rich programming is programming where one or more information spaces are integral to the operation of the programs being constructed.

An *information space schema* is a (often formal) structure that characterizes the common names, shapes, operations and constraints for an external information space.

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. This form must reflect the information space schema of the information space in the strongly-typed representation on the programming language side.

A *component signature* is the signature of software component or information space 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.

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.

2. Themes and Challenges

In this section, we state the primary themes and challenges we have encountered for strongly-typed information-rich functional programming for internet scale information spaces. We refer the reader to [8] for more detailed descriptions.

Theme: 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. 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).

Theme: Few existing programming languages are able to seamlessly integrate external internet-scale information sources in a scalable way. A repeated surprise in this work has been that almost no existing languages have scalable architectures for the direct integration of stable external information spaces as strongly-typed components. Existing language/tool architectures generally use code generation or eager macros, techniques which scale poorly because they are not on-demand.

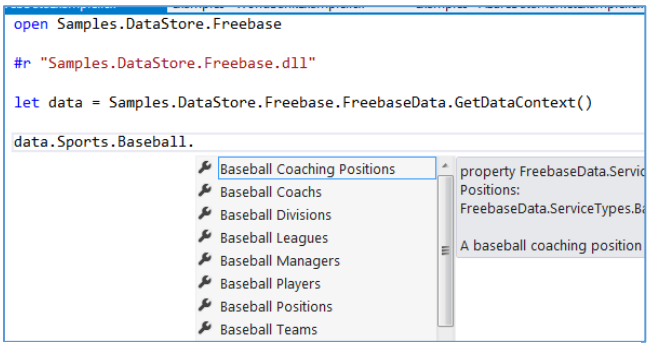

Figure 1. Exploring the Baseball Domain on Freebase

The current industry alternative to static type bridging is to use dynamically typed information representations (often in a dynamically typed language). This scales well but discards the benefits of strongly-typed programming. This 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.

Theme: 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 (SQL, XML, Web Services, CORBA, DCOM, Linked Data, OData, GData, Atom, REST, RSS, JSON, RDF, Protocol Buffers…). Our natural instinct as computer scientists is to seek a single, unifying transport standard for data sources, and the history of software is littered with such attempts. In the end technologies often trend towards lowest-common-denominator approaches such as XML or JSON. Our programming languages need information integration architectures that are open, rather than tied to these representations. A provider model avoids the lowest common denominator by opening the programming language to work with multiple standards in a scalable way.

Theme: A strongly-typed, functional-first language is an excellent starting point for strongly-typed information-rich programming. This is for these reasons:

- *Type inference.* F# and all strongly-typed functional languages use type-inference extensively, requiring considerably fewer type annotations than C#, C++ or Scala, and many fewer than Java. For example, type annotations are not needed on return types for F# functions, and are rarely needed at value declarations. This is particularly crucial when dealing with provided information spaces that contain thousands or millions or types, as these types rarely need to be explicitly named, and instead flow from provided values through the code.
- *The Best of OO, the Best of Functional.* Mixed functional/OO languages use functional constructs for data manipulation and OO for data representation. OO techniques are unrivalled for representing large library designs, so it makes sense that they are used for this role when representing even larger information spaces. Functional techniques are unrivalled for data transformation, query languages and compositional control abstractions. For examples, provided collections use functional abstractions such as sequences, LINQ queries [5], and the F# asynchronous programming model [9].
- *Interactive Execution.* The combination of type providers with a REPL (in our case F# Interactive) gives a code-focused, data-scripting experience for tasks using large data sources.
- *Prior data paucity.* Functional programming languages have historically been data-deficient, in the sense that accessing external data sources has been hard, partly due to the lack of libraries, and also due to the philosophical roots these languages have in elegant closed-world mathematical systems. The needy feel the benefit of type providers very strongly!
- *Meta-programming foundations.* At the API level, F# type providers utilize several .NET and F# meta-programming idi-

oms, including F# quotations [6]. The pre-existence of these facilities was an important factor in making the type provider implementation possible within reasonable resource constraints.

Theme: Design-time assistance. Modern development environments use types to simplify the implementation of "design time" tooling, including Interactive type checking during development ("red squigglies"), provision of context-sensitive declaration lists ("auto-completion"), type-directed information and documentation on gestures such as mouse-hover ("quick info"), type and name-directed help systems ("F1 help"), and safe refactorings.

Some design choices of F# type providers are very much driven with the design-time experience in mind. The value of F# type providers lies very much in being able to use design-time tooling to navigate and explore information spaces. Design-time assistance also has a strong influence on information-space design, as much of the time in developing a type provider is spent in improving the usability of the completion lists offered to the data engineer. Further, individual providers sometimes show sample data in provided documentation, and completion-list filtering provides a simple search mechanism.

Theme: Connected programming. "Connected programming" is where connectivity to remote information sources is required during program development and/or execution. Connected programming is a common theme in language-integration of large information sources because these sources are normally (though not always) hosted remotely.

In the case of this work, connected programming means we can assume type providers can access live data sources at design-time, giving the developer an up-to-date schematization during development.

In many 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. 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.

Theme: Bringing Simplicity and Consistency across Information Spaces. One aim of F# type providers is that data/information programming experience is consistent across different data services. This can be seen from the seemingly disparate examples in [8]. 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 [7].

Theme/Challenge: 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 results of erasure functions)

We note that there is a strong trend to towards stable, rich information sources delivered through the internet [8]. Further, in typical enterprises there are many information sources with relatively stable schemas that make up the "information base" of a modern 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. So, while information sources change, in practice a growing number of information sources don't change quite as much as one might think.

Type providers address some aspect of schema changes in these 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 auto-complete 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
- Deprecation ("obsolete") attributes can be propagated from external information sources. If a continuous integration build process (e.g. nightly) is used, then schema change or deprecation is detected as soon as a typecheck is performed against the modified schema. This potentially allows for early detection and remediation of problems arising from schema change.

We 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. 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. [2, 3, 1].

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. 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 may be relevant here [1].

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 [5], 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.

Theme: Security boundaries in schema information. Connected programming introduces the themes of security and authentication when accessing both schema (at design-time) and data (at runtime). For most of the work described in [8] we 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. However, this assumption breaks down for many interesting and reasonable applications of type-bridging mechanisms (imagine, for example, a type provider providing information from a multi-security-level classified data source). Our view of compiler/tool architecture and of "what is a program" may need a radical overhaul once we acknowledge that even writing and checking a program may require a range of credentials.

Theme: Granularity of Schematization. A common theme in accessing data in a strongly-typed way relates to the granularity of the schematization of the data. The question is not just "schematized" v. "unschematized" but rather "how much schema"? For example, [8] describes a bespoke type provider for World Bank data, where schematization is at the granularity of individual countries and indicators, where the Azure Data Market version of this provider applies a much coarser schematization, where individual country and indicator names are not part of the schema.

Individual providers can provide multiple granularities over the same data sources – this is 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 thought and design. This is especially true given the new power that scalable type spaces introduce.

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 [4], so it makes sense to propagate this information in the types provided by a type provider. Providing further metadata and constraints into the type system is an important open question and direction for future research.

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 F# compiler for on-demand 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.

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 there may be social or interoperability reasons for doing so.

3. Summary and Future Directions

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

Automated type bridging mechanisms radically expand the role for names and types**.** In the examples explored in [8], types are used much more extensively than in the equivalent loosely-typed information access using XML, strings or JSON. Expanding the role of types brings many benefits, but challenges existing, comfortable assumptions about what types are, what role they serve, what soundness result they guarantee, how they are selected and what temporal properties they should have.

Many future opportunities exist for applying strong typing when interacting with external information spaces. Some of these avenues are implied by addressing the themes and challenges outlined in this paper. Some avenues represent new ways to think about the interplay between language and tools architecture: for example, providers could give search functionality, recommendations, sample data or metadata-edit functionality for the provided metadata space.

References

[1] G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental Typechecking for Class Upgrades. In *ECOOP*, pages 235–259. 2008.

[2] S. Drossopoulou, D. Wragg, and S. Eisenbach. What is Java binary compatibility? OOPSLA '98, pages 341–361. ACM, 1998.

[3] S. Eisenbach and C. Sadler. Changing Java Programs. In *ICSM 2001*, Florence, Italy, November 2001.

[4] A. 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.

[5] E. Meijer, B. Beckman, and G. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Int. ACM Conf. on Mgmt. of Data*. ACM, 2006.

[6] D. 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.

[7] D. Syme and K. 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 Aug 2012.

[8] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F#3.0 - Strongly-Typed Language Support for Internet-Scale Information Sources. Technical Report MSR-TR-2012-101, Microsoft Research, 2012.

[9] D. Syme, T. Petricek, and D. Lomov. The F# asynchronous programming model. In *Proceedings of the 13th international conference on Practical Aspects of Declarative Languages*, PADL'11, pagpages 175–189, Berlin, Heidelberg, 2011. Springer-Verlag.