acmqueue

Why LINQ Matters: Cloud Composability Guaranteed

The benefits of composability are becoming clear in software engineering.

Brian Beckman, Microsoft Corporation

Composability is an aspect of component design that addresses the freedom to select and combine generic components in nearly arbitrary configurations to support a wide variety of applications, even applications that were not anticipated by the designers of the components. For example, electrical components are routinely designed so that switches, junctions, and loads can be configured in almost any order to fulfill an enormous variety of applications. Component designers do not need to know the specifics of an application, and component users are not overly constrained by arbitrary choices made by designers. Similarly, in mechanical engineering many applications can be built entirely of generic brackets, hinges, fasteners, and so on, designed at the outset to be configured in any order and layout made possible by standardization of sizes, screw pitches, strength-of-materials, etc.

In this article we use LINQ (Language-integrated Query) as the guiding example of composability. LINQ is a specification of higher-order operators designed specifically to be composable. This specification is broadly applicable over anything that fits a loose definition of “collection,” from objects in memory to asynchronous data streams to resources distributed in the cloud. With such a design, developers build up complexity by chaining together transforms and filters in various orders and by nesting the chains—that is, by building expression trees of operators.

Encoding and transmitting such trees of operators across tiers of a distributed system have many specific benefits, most notably:

- Bandwidth savings from injecting filters closer to producers of data and streams, avoiding transmission of unwanted data back to consumers.
- Computational efficiency from performing calculations in the cloud, where available computing power is much greater than in clients.
- Programmability from offering generic transform and filter services to data consumers, avoiding the need for clairvoyant precanning of queries and data models at data-producer sites.

This article also covers lifting composability from programs into the cloud via REST (representational state transfer), and mapping between expressions in programs and resource specifications in URIs (uniform resource identifiers). It also addresses the subtle hazards of designing for composability: seemingly unimportant, arbitrary choices can render a design fundamentally uncomposable.

META-DESIGN PRINCIPLE

Designing for composability is not *merely* separating interface and implementation. It also means a certain uniformity in the interfaces: *a meta-design principle*. It wouldn't do much good if every electrical socket-and-plug design were of a custom and idiosyncratic shape, even if the design were sufficient to carry the required current. It's precisely because sockets and plugs are standardized in

shape that we get the flexibility to combine them in nearly arbitrary configurations.

The physical engineering disciplines (e.g., electrical and mechanical) have a well-established history of designing for composability because the benefits are so obvious and overwhelming, but the benefits are becoming increasingly clear in software engineering as well. Software designers have long accepted blackboxing: the need to encapsulate implementation details inside interfaces that expose minimal, precise, complete contracts. Designers are now beginning to abstract over the interfaces themselves and to realize the combinatorial benefits of composability.

FUNCTIONS AS COMPOSABILITY ATOMS

The discipline of pure functional programming, exemplified by the programming language Haskell (<http://www.haskell.org>) and its predecessor Miranda ([http://en.wikipedia.org/wiki/Miranda_\(programming_language\)](http://en.wikipedia.org/wiki/Miranda_(programming_language))), brought the absolute composability guarantees of mathematical functions into the world of programming. Their influence, combined with imperative and object-oriented programming traditions, represents a recent, deep, and important consolidation: one big concept, *function*, incorporates several smaller concepts as cases. Relations, objects, states, and streams all have natural representations in functional style. The increasing favor of this style is evidenced by the following:

- The incorporation of higher-order functions (http://en.wikipedia.org/wiki/First-class_function; http://en.wikipedia.org/wiki/Higher-order_function) for transforming, filtering, and aggregating data into workaday imperative programming languages, most notably C# and JavaScript.
- The general recognition of immutability as a benefit, especially in concurrent and distributed systems; see Eric Lippert's comments (<http://blogs.msdn.com/b/ericlippert/archive/2007/11/13/immutability-in-c-part-one-kinds-of-immutability.aspx>) and a relevant thread in the Python users' forum (http://groups.google.com/group/comp.lang.python/browse_thread/thread/29c62cbee7a6b598/df5b676f6f695eb9).
- The spread of libraries such as LINQ (http://en.wikipedia.org/wiki/Language_Integrated_Query) and underscore.js (<https://github.com/documentcloud/underscore>).
- Whole languages such as Scala, F#, and Clojure.
- Language overlays such as coffee-script (<https://github.com/jashkenas/coffee-script>).

LINQ AS A MULTIDOMAIN COMPOSABILITY PATTERN

LINQ is a case study in taking the composability of functions one step further by specifying a collection of composable higher-order functions—the SQOs (standard query operators; http://en.wikipedia.org/wiki/Language_Integrated_Query#Standard_Query_Operators). This specific design generalizes over a loose abstraction of a collection and covers a comprehensive variety of domains:

- Data structures in memory (lists, trees, graphs, queues...).
- Tables in a database (even with primary- and foreign-key constraints).
- Asynchronous data streams (RX; <http://msdn.microsoft.com/en-us/data/gg577609>).
- Slash-separated terms in URIs; thus, resources in the cloud.
- Signal-processing primitives (e.g., convolutions and Fourier transforms).
- Coded DOMs (code document object models); expression trees; ASTs (abstract syntax trees)—that is, over programming languages themselves.
- HTML, XML, and JSON (JavaScript Object Notation) documents.

- Continuations, exceptions, alternatives, and more.

This design brings the same thorough collection of composable operators to all these domains. You could say that LINQ brings *composability itself* to these domains. If you implement the SQOs, then you are guaranteed composability. LINQ wasn't the first and certainly isn't the only way to do it, but it's exemplary enough to support the other observations in this article, namely:

- Lifting composability in programs to composability in the cloud.
- Illustrating the hazards facing would-be composable designs.

FROM PROGRAMS TO THE CLOUD

LINQ's SQOs, then, serve as our exemplar of the composability-in-itself meta-design principle. As typically implemented in programs, the SQOs apply functions to collections in various ways, recognizing *functions as the natural interaction convention between composable components in programs*.

Leaping from composability within programs to composability in the cloud, you must "implement" SQOs over *the natural interaction convention between composable components in the cloud*, whatever that may be. This is just another domain for LINQ, viewed as a specification for composable operators over anything that satisfies LINQ's loose abstraction of a collection. The cloud is just a loose collection of resources specified by URIs.

What is "the natural interaction convention between composable components in the cloud"? Perhaps the most important such interaction convention today is REST (<http://en.wikipedia.org/wiki/REST>), in which client and server interact via HTTP, by opaque URIs that represent *resources*. REST specifies no semantics on URIs, only a very light postfix syntax of slash-separated terms. Protocols such as OData (<http://www.odata.org/>) use URI encoding and REST to provide data frameworks to the cloud, specifically by empowering a client to specify the result of a desired computation or query as a resource. To this let's add the following ideas:

- **RADICAL COMPOSABILITY.** Users build complexity via chaining sequences of SQO expressions in arbitrary order rather than via a rich syntax over a presumed data model, as in OData.
- **BIDIRECTIONAL MAPPING.** This is mapping between expression chains in programs and resource specifications in URIs. Given such a mapping, you may reason over expressions in programs and resources in cloud interactions as if they were the same.
- **INJECTABILITY.** The receiver of an expression chain may maintain standing operations over multiple communications so as to save bandwidth. A client requesting, say, a stream of "Chinese restaurants near me," may inject a filter one time to the server, which then avoids sending other irrelevant business records to the client over future sequences of push notifications. A server, flooded by too-frequent location updates from a client, may inject a filter into the client that says "only every tenth update," which the client applies just before performing physical communication.
- **BROAD APPLICABILITY.** The same composable design works over static data resources and time-dependent, asynchronous data streams (and other domains).

At this point, assume a natural correspondence between composable operator-chain expressions in programs and composable resource specifications in the cloud. A specific way to mechanize that correspondence is presented in the following sections, but the correspondence itself allows you to reason the same way over what works in programs and what works in the cloud.

EMBEDDING LINQ IN URI SYNTAX

The typical implementation of LINQ is embedded in a programming language that supports higher-order functions. Adopting LINQ, however, only means accepting the specification of the SQOs. LINQ, as a design, does not need an ordinary programming language at all. By noting that (1) a nested tree of operators can be converted to a pure-postfix *chain* of operators; and (2) a chain of operators separated by dots in a programming language is formally isomorphic to a chain of terms separated by slashes in a URI, you can embed any LINQ chain in URI syntax and use HTTP as an “expression-embedding language” in RESTful Web services.

For example, imagine the following expression, in pseudo-C#, that represents the phone numbers of customers with recent purchases:

```
Customers
    .Where(customer =>
        (DateTime.Now - customer.Orders.Last().dateTime)
        < TimeSpan.FromDays(365))
    .Select(customer => customer.PrimaryContactPhone)
    ;
```

This says, “Keep the customers *Where* the difference between the date and time now and the date and time of the last purchase is fewer than 365 days, then *Select* the primary phone contact number from each customer in the resulting collection of customers.” *Where* and *Select* are both LINQ SQOs. Each one takes two arguments—a data collection argument to the left of each dot and a higher-order function inside parentheses—and each produces a new collection. That’s the essence of their chain composability: each SQO expression represents a data collection ready to be dotted into the next SQO down the chain in left-to-right order.

The higher-order-function argument of the *Where* SQO is a predicate written as a lambda expression:

```
customer => ...function-body expression depending on customer...
```

Read this as “the function of *customer* that performs the computation specified by the function-body expression and produces the resulting value of the expression.” Since this particular lambda expression represents a predicate, it should produce a Boolean value. In JavaScript, you would write exactly the same thing as

```
function(customer) {return ...function-body expression...;}
```

The higher-order-function argument of the *Select* SQO is another lambda expression that just picks out the phone-number field from a customer record, but it could do arbitrary computation.

Most languages or libraries call this *Select* operator *map*, but LINQ’s designers chose the name *Select* to appeal to SQL developers. In some theoretical discussions, the operator is also called *Project*. Let’s stick with *Select* for now.

Since the expression chain contains no nested operators, it’s already in *shallow postfix* form if we consider the lambda expressions atomic. This shallow postfix form is

```
Customers.Where(...LambdaW...).Select(...LambdaS...)
```

Just prepend a protocol and domain, and replace the dots with slashes:

```
https://myQueries.com/Customers/Where(...LambdaW...)/Select(...LambdaS...)
```

Then URL-encode the lambdas, ending up with something like this:

```
https://myQueries.com/Customers/Where(customer%3D%3E(DateTime.Now-customer.
Orders.Last().dateTime)%3CTimeSpan.FromDays(365))/Select(customer%3D%3Ecustomer.
PrimaryContactPhone)
```

Thus, you have a representation of the entire expression in a URI. A RESTful server can interpret expressions such as this in a security sandbox and provide a very general query service without having built-in, precanned queries.

If you cannot or do not want to consider the lambdas as atomic, then you can go all the way to *deep postfix* form, writing the entire AST in postfix and encoding it in a URI.

You can do this in two stages: first keep the lambdas imploded so you can see the difference between deep prefix and shallow prefix; then explode the lambdas. Begin by rewriting the query first in prefix form, dragging everything that was on the left of a query-operator dot over to the right inside the parentheses of the operator in first position. This turns the expression inside out like the sleeve of a sweater, as follows:

```
Select(Where(Customers, ...LambdaW...), ...LambdaS...)
```

Now, it is easy to see that each level is a binary operator with a data collection in the first argument slot and a lambda in the second slot. This corresponds to the AST in figure 1. Without

FIGURE 1 AST Showing Imploded Lambdas

exploding the lambdas. from there, left-to-right, depth-first traversal yields a deep postfix form, still with lambdas imploded:

<https://myQueries.com/Customers/2.LambdaW/Where/4.LambdaS/Select>

FIGURE 2 AST with Exploded Lambdas

It looks similar to the shallow postfix form, just with the SQOs *post-lambda*, as it were. The SQOs appear after the lambdas instead of hosting lambdas inside their parentheses. In deep postfix form, there are *never* any parentheses, and that's the whole point. This should remind you of PostScript or Forth or RPN (reverse Polish notation) calculators, because they are all deep postfix (http://en.wikipedia.org/wiki/Concatenative_programming_language).

Now, explode out the lambdas into the AST shown in figure 2. With no parentheses, its RESTful encoding in a URI is trivial:

```
https://myQueries.com/Customers/DateTime.Now/customer/.Orders/Last/.dateTime/-/365/
Const/TimeSpan.FromDays/Lt/lambda.customer/Where/customer/.PrimaryContactPhone/
lambda.customer/Select
```

The remaining dots denote property-accessor calls, system calls, or lambda-variable declarations, as opposed to SQO calls. URI encoding replaced all dots preceding SQO calls with slashes.

Note that there are other options for encoding lambdas in postfix. This example uses a representation in which variables appear before they're declared. This requires an execution model that saves variables and dependent operations symbolically on the stack, to be resolved later when the lambda term declaring the variable arrives. Other competent choices abound. For example, PostScript and Factor encode lambda expressions in arrays where the variable declarations precede the function bodies. This amounts to a cheater prefix notation embedded in otherwise postfix languages.

We've departed the realm of human-readable syntax (except for lovers of Forth) but have found a URI encoding of ASTs that's trivial for machines to read and write. We probably don't need to go quite this far, even for nested LINQ. Parenthesis counting can just as easily manage nesting while retaining human readability.

For an example with nested SQOs, imagine the following expression, again in pseudo-C#, that represents the line items for high-value orders from a list of customers:

```
Customers
    .Where(customer => customer.Orders
        .SelectMany(order => order.LineItems)
        .Sum() > 1000)
    .SelectMany(customer => customer.Orders)
    .SelectMany(order => order.LineItems)
    ;
```

Save some ink by replacing lambda variable *customer* with the shorter *c* and lambda variable *order* with the shorter *o*. You can write the URI form without hesitation:

```
https://myQueries.com/Customers/Where(c%3D%3Ec.Orders/SelectMany(o%3D%3Eo.Line-
eItems)/Sum(%)%3E1000)/SelectMany(c%3D%3Ec.Orders)/SelectMany(o%3D%3Eo.LineItems)
```

Then you can convince yourself that parenthesis counting reveals the nesting in this and every case.

HAZARDS IN DESIGNING FOR COMPOSABILITY

While LINQ is a fine example of a composable design that works equally as well over objects in memory as over resources in the cloud, it's certainly not the only one. Designers who embrace radical composability, however, will confront some hazards. If they get something just slightly wrong in the design, then users can no longer specify what they need within the design.

In a typical example, suppose you need a collection of all orders from big-spending customers. Your library provides a way to filter out low-spending customers, and it provides a *composable* way to get the orders for each of those customers. So far, so good. In pseudo-C#, you might try:

```
Customers
    .Where(customer => customer.TotalSpending() > 1000)
    .Select(customer => customer.Orders)
;
```

This does not yield the required collection of orders, however; instead it yields a collection of *collections* of orders—one internal collection of orders for each customer. You need to flatten the top-level collection. You must leave the library and write custom code.

In a cloud setting, leaving the library means proliferating custom code for every uncovered case, restricting the ability to build novel expressions just from composable building blocks. Anywhere you have an expression *like* the one above, you must do something out-of-band to remove the unwanted extra structure.

In a program setting, leaving the library means risking insertion of brittle workaround code in various places in the system. Perhaps the programmer knows that `Orders` are arrays, so writes block-copy operations to flatten them. Later, when `Orders` changes implementation from array to gzipped XML, this code fails unexpectedly.

If, however, the library had provided the required `SelectMany` in the first place, then the programmer would have written the following, which has only one difference from the previous code (underlined):

```
Customers
    .Where(customer => customer.TotalSpending() > 1000)
    .SelectMany(customer => customer.Orders)
;
```

BREADTH AND DEPTH OF COMPOSABILITY ITSELF

It's somewhat amazing that a single specification of operators such as LINQ's SQOs applies equally as well to asynchronous data streams as to objects in memory—but *it does*. Users can specify arbitrary transformations not only on pulled data resources, but also on pushed asynchronous data streams using the same set of SQOs. In both cases, expression components can be transparently injected upstream to the data-producer machines, whether client or server, transforming and filtering data at the headwaters for bandwidth savings and reduction of “chattiness.”

In a kind of self-referential joke, however, the same set of SQOs that works over functions in a program works over resources distributed in the cloud. Thus, there is an octuplet application of the same SQO design:

- SQOs operating on data within programs or on data over the cloud.
- Data distributed in space or data distributed in time.
- Transforms and filters executed at the data-producer site versus at the data-consumer site, respecting concerns such as bandwidth, latency, and security.

LINQ borrowed its essential concepts once again from Haskell, specifically from its initialization library called the Prelude. The essence of the technique is *a loose abstraction of a collection* of things—for example (don't yawn), customers, orders, items. Each customer has a number of orders, and each order has a number of items. From the point of view of the pattern, it does not matter whether these customers are in memory, or delivered *n* at a time to callbacks, or threaded through continuations, or in offline document stores accessible by Web-service calls, etc. What matters is just that there is an abstraction representing them somewhere in your system where you want to manipulate them. The required composable operators fall into categories (with examples from LINQ and the Haskell Prelude):

- **INSERT** elements into collections (e.g., `new List(...)` / `return`). This is how you get elements into a collection. For example, convert a customer record into a one-record table; make a singleton list (containing a single customer object).
- **TRANSFORM** elements and produce a new collection (e.g., `.Select` / `map`). For example: produce a list of customers' primary phone numbers from a list of customers, one phone number for each customer.
- **FILTER** elements out of the collection based on a predicate (e.g., `.Where` / `filter`). For example, produce a list of customers with big orders; there will be fewer of these, in general, than in the original collection.
- **EXPAND** elements from one collection into new subcollections and combine or concatenate the new collections (e.g., `.SelectMany` / `concatMap`, `bind`, `>>=`). For example, get all first-degree friends of the customers; get all orders from all customers; get all line items from all orders.
- **AGGREGATE** elements and produce a result that depends on *all* the elements (e.g., `.Aggregate` / `fold`; and `.Take` and `.Skip` / `take`, `drop` also fall in this category). This is how you get elements out of the collection (technically, this is a co-monadic operator, EXTRACT). For example, get the average spending over all customers.

These are fundamental. If a collection abstraction supports appropriate primitives in each of these categories, then composability is guaranteed.

HOW TO RUIN A LIBRARY

You don't have to write *exactly* the LINQ SQOs; there is plenty of freedom if you understand the fundamentals. One big way to blow it, however, is to overlook the EXPAND category, which designers often do because it doesn't fit the map/filter/fold mantra familiar in some quarters. The EXPAND category is essential; in fact, it's the most important one. Without it, the library can't represent the most general kind of collection and thus can't be extended to new settings such as asynchronous data streams or resources in the cloud. With it (and with INSERT), all the other categories can be simulated exactly (for more information, see <http://channel9.msdn.com/Shows/Going+Deep/Bart-De-Smet-MinLINQ-The-Essence-of-LINQ>).

The diagrams in figure 3 demonstrate the categories of operator. Together, the diagrams show why the EXPAND category is most essential. It's the necessary twin of FILTER, even though it's more general.

FIGURE 3

Categories of Composable Operators

Another way to ruin a library is to get the order of arguments wrong. To chain operators, let alone to embed them in URIs and REST, you want the collection argument in first position, always. Traditionally, dialects of Lisp put the *function* argument first, at least for `map`, because it makes expressions read, “Map this function over that collection.” Library designers with Lisp backgrounds might just reflexively write `map` that way. This minor syntactic lapse means that `map`, a.k.a. `Select`, disrupts the pattern and cannot be composed in dotted chains except at the beginning. For example, suppose you want a sorted price list for a certain digital camera from a collection of vendors:

```
CameraVendors
    .Where(vendor => vendor.Offers.Contains(mySelectedCamera))
    .Select(vendor => vendor.Offers
        .FirstWhere(offer => offer.Item == mySelectedCamera))
    .OrderBy(offer => offer.Price)
    .ForEach(offer => offer.Print())
;
```

Suppose you modify the expression to reconsider prices after shipping costs. Just compositionally

insert another `Select` (shown underlined) in the middle of the chain, leaving everything else the same:

```
CameraVendors
    .Where(vendor => vendor.Offers.Contains(mySelectedCamera))
    .Select(vendor => vendor.Offers
        .FirstWhere(offer => offer.Item == mySelectedCamera))
    .Select(offer =>
        offer.Price + offer.Shipping(myZipCode, myShippingOptions))
    .OrderBy(offer => offer.Price)
    .ForEach(offer => offer.Print())
;
```

Now, suppose `Select` were designed traditionally, so that it takes its collection argument in second position rather than to the left of the dot, which is effectively the first position. You would start with something like this:

```
SelectUgly(vendor => vendor.Offers
    .FirstWhere(offer => offer.Item == mySelectedCamera),
    CameraVendors)
    .OrderBy(offer => offer.Price)
    .ForEach(offer => offer.Print())
;
```

Adding the transformation that considers shipping costs requires major syntactic surgery along the following lines, adding the underlined text and removing the struck-through text:

```
SelectUgly(offer =>
    offer.Price + offer.Shipping(myZipCode, myShippingOptions),
    SelectUgly(vendor => vendor.Offers
        .FirstWhere(offer => offer.Item == mySelectedCamera),
        CameraVendors)=
    .OrderBy(offer => offer.Price)
    .ForEach(offer => offer.Print())
)
    .OrderBy(offer => offer.Price)
    .ForEach(offer => offer.Print())
;
```

This completely disrupts the compositional structure. Instead of just inserting a line in the appropriate place, you must splay (un-nest and re-nest) the expression. The order of arguments might seem like a small point, but if you get it wrong, it ruins exactly the composability desired.

CONCLUSION

LINQ's pattern is bulletproof and pervasive, applying to a surprisingly broad array of software domains such as asynchronous streams, stateful computations, I/O, exceptions, alternatives—anything that fits a loose abstraction of “collection of things.” The pattern is always composable in *expression trees*, where dependent operators follow each other in nested sequences. It can be encoded equally as well in ordinary programming language syntax as in pure-postfix notations such as URIs.

Because the cloud fits the loose definition of a “collection of things,” LINQ’s pattern of composable transforms covers distributed resources in the cloud as well as it covers locally addressable resources in programs. Packaging subexpressions and injecting them closer to data-production sites allows you to gain bandwidth, chattiness, and security benefits in RESTful Web services.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BRIAN BECKMAN, works in Bing on Maps and Signals. He has held many positions at Microsoft since 1992, from Crypto (SET) to Biztalk to research in functional programming. He wrote the first version of the Time Warp Operating System on the Caltech Hypercube 1984-1989. He holds a Ph.D. in astrophysics from Princeton University and has filed more than 80 patents, with 27 issued.

© 2012 ACM 1542-7730/12/0200 \$10.00