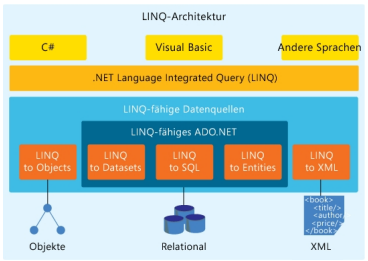
Einführung

Anson Horton ist bereits fast sechs Jahre lang als Programmmanager bei Microsoft tätig. Er war von Anfang an beim C#-Team dabei. Davor arbeitete er im C++-Team. Anson Horton hat am Entwerfen der C#-Sprache und des Compilers, am C#-Projektsystem, an der C#-IDE (IntelliSense) sowie an der Auswertung und Problembehandlung des C#-Ausdrucks mitgewirkt. Er führt einen Blog unter blogs.msdn.com/ansonh , den er möglichst selten aktualisiert. Der folgende Artikel ist eine persönliche Abhandlung.

Ich war ein großer Anhänger der Discovery Channel-Serie „Connections“, die von James Burke ins Leben gerufen und moderiert wurde. Die grundsätzliche Annahme dieser Serie war, dass scheinbar zusammenhanglose Erfindungen und Entdeckungen wiederum andere Erfindungen beeinflusst haben, was uns letzten Endes viele moderne Annehmlichkeiten beschert hat. Der Gedanke dabei ist der, dass Fortschritt nicht isoliert in einem Vakuum erfolgt. Es verwundert nicht, dass dies ebenso auf Language Integrated Query (LINQ) zutrifft. Einfach ausgedrückt, handelt es sich bei LINQ um eine Serie von Spracherweiterungen, die Daten in einer typenzuverlässigen Weise unterstützt. LINQ wird mit der nächsten Version von Visual Studio (Codename „Orcas“) veröffentlicht. Die abzufragenden Daten können im Format XML (LINQ zu XML), Datenbanken (LINQ-aktiviertes ADO.NET, das LINQ-to-SQL, LINQ-to-Dataset und LINQ-to-Entities einschließt), Objekte (LINQ-to-Objects) und so weiter vorliegen. Die LINQ-Architektur wird in Abbildung 1 dargestellt.



Im Folgenden werden einige Codes untersucht. Ein Beispiel für eine LINQ-Abfrage in der bevorstehenden „Orcas“-Version von C# könnte wie folgt aussehen:

[code xml:lang="csharp"]var overdrawnQuery = from account in db.Accounts
where account.Balance Wenn die Ergebnisse dieser Abfrage unter Verwendung von „foreach“ durchlaufen werden, besteht jedes zurückgegebene Element aus einem Namen und einer Adresse eines Kontos, das einen Stand von weniger als 0 hat.

Aus dem obigen Beispiel ist sofort ersichtlich, dass die Syntax SQL ähnelt. Vor mehreren Jahren dachten Anders Hejlsberg (leitender Designer von C#) und Peter Golde daran, C# zu erweitern, um die Integration der Datenabfrage zu optimieren. Peter Golde, der zu der Zeit die C#-Compilerentwicklung leitete, untersuchte die Möglichkeit zur Erweiterung des C#-Compilers,

insbesondere zur Unterstützung von Add-Ins, die die Syntax domänenspezifischer Sprachen wie SQL überprüfen konnten. Anders Hejlsberg dagegen wollte eine tiefere, spezifischere Integrationsebene konzipieren. Er stellte sich einen Satz von „Sequenzoperatoren“ vor, der mit einer beliebigen IEnumerable implementierenden Sammlung funktionieren würde, und an Remoteabfragen für Typen, die IQueryable implementierten. Letzten Endes erhielt die Idee der Sequenzoperatoren die meiste Zustimmung, und Anfang 2004 reichte Anders Hejlsberg für die von Bill Gates regelmäßig unternommene „Think Week“ einen Bericht darüber ein. Die Resonanz war überwältigend positiv. In den Anfangsphasen des Entwurfs hatte eine einfache Abfrage die folgende Syntax:

[code xml:lang="csharp"]sequence locals = customers.where(ZipCode == 98112);[/code]

In diesem Fall war die Sequenz ein Alias für IEnumerable, und das Wort „where“ ein besonderer Operator, der vom Compiler verstanden wurde. Die Implementierung des Where-Operators stellte eine reguläre statische C#-Methode dar, die einen Prädikatsdelegaten aufnahm (d. h. einen Delegaten in der Form von bool Pred(T item)). Die Idee war, dass der Compiler ein besonderes Wissen über den Operator haben sollte. Dadurch wäre der Compiler in der Lage, die statische Methode korrekt aufzurufen und den Code zu erstellen, um den Delegaten mit dem Ausdruck zu verknüpfen.

Angenommen, das obige Beispiel wäre die ideale Syntax für eine Abfrage in C#. Wie würde diese Abfrage in C# 2.0, ohne jegliche Spracherweiterungen, aussehen?

[code xml:lang="csharp"]IEnumerable locals = EnumerableExtensions.Where(customers,
delegate(Customer c) { return c.ZipCode == 98112;
});[/code]

Dieser Code ist viel zu ausführlich, und schlimmer noch, er erfordert beachtlichen Tiefgang, um den relevanten Filter (ZipCode == 98112) zu finden. Dieses Beispiel ist noch recht einfach. Stellen Sie sich vor, wie unlesbar alles bei mehreren Filtern, Entwürfen und so weiter wäre. Der Grund für die Ausführlichkeit liegt in der Syntax, die für anonyme Methoden erforderlich ist. Bei einer idealen Abfrage wäre für den Ausdruck lediglich die Auswertung des Ausdrucks erforderlich. Der Compiler versucht dann, den Kontext abzuleiten, zum Beispiel, dass sich ZipCode wirklich auf den ZipCode bezieht, der unter Customer definiert wurde. Wie kann dieses Problem behoben werden? Das Wissen bestimmter Operatoren in die Sprache hartzucodieren kam beim Entwurfsteam nicht gut an, daher musste eine andere Syntax für anonyme Methoden gefunden werden. Diese sollte äußerst präzise sein, dabei jedoch nicht unbedingt mehr Wissen benötigen, als der Compiler derzeit für anonyme Methoden erfordert. Letzten Endes hat das Team Lambda-Ausdrücke ausgearbeitet.

Lambda-Ausdrücke

Bei Lambda-Ausdrücken handelt es sich um ein Sprachfeature, das in vielerlei Hinsicht

Ähnlichkeit mit anonymen Methoden hat. Wenn Lambda-Ausdrücke schon von Anfang an in die Sprache eingeflossen wären, hätte es eigentlich gar keinen Bedarf für anonyme Methoden gegeben. Die Idee ist grundsätzlich die, dass Code wie Daten behandelt werden kann. In C# 1.0 ist es üblich, Zeichenfolgen, Ganzzahlen, Verweistypen usw. an Methoden zu übergeben, damit die Methoden anhand dieser Werte vorgehen können. Durch anonyme Methoden und Lambda-Ausdrücke wird der Wertebereich auf Codeblöcke erweitert. Dieses Konzept ist in der Funktionsprogrammierung üblich.

Ersetzen Sie z. B. beim oben genannten Beispiel die anonyme Methode mit einem Lambda-Ausdruck:

[code xml:lang="csharp"]IEnumerable locals = EnumerableExtensions.Where(customers, c => c.ZipCode == 91822);[/code]

Dabei fallen mehrere Dinge auf. Zum einen kann die Kürze des Lambda-Ausdrucks einer Anzahl von Faktoren zugeschrieben werden. Zunächst einmal wird das Konstrukt nicht mit dem Delegatenschlüsselwort eingeführt. Stattdessen teilt ein neuer Operator, =>, dem Compiler mit, dass es sich hier nicht um einen regulären Ausdruck handelt. Zweitens wird der Customer-Typ von der Verwendung abgeleitet. In diesem Fall sieht die Signatur der Where-Methode etwa folgendermaßen aus:

[code xml:lang="csharp"]public static IEnumerable Where(IEnumerable items, Func predicate)[/code]

Der Compiler kann ableiten, dass sich „c“ auf einen Kunden bezieht, da der erste Parameter der Where-Methode IEnumerable lautet, sodass T dementsprechend Customer sein muss. Mit diesem Wissen überprüft der Compiler außerdem, dass Customer ein ZipCode-Mitglied hat. Zum Schluss ist noch zu erwähnen, dass kein Rückgabeschlüsselwort angegeben ist. In der syntaktischen Form wurde das Rückgabemitglied ausgelassen, doch der Grund dafür ist lediglich syntaktische Bequemlichkeit. Das Ergebnis des Ausdrucks wird nach wie vor als der Rückgabewert angesehen.

Genau wie anonyme Methoden unterstützen Lambda-Ausdrücke die Erfassung von Variablen. Zum Beispiel ist es möglich, auf die Parameter oder lokalen Variablen der Methode zu verweisen, die den Lambda-Ausdruck innerhalb des Lambda-Ausdruckstexts enthält:

[code xml:lang="csharp"]public IEnumerable LocalCusts(IEnumerable customers, int zipCode) { return EnumerableExtensions.Where(customers, c => c.ZipCode == zipCode); }[/code]

Schließlich unterstützen Lambda-Ausdrücke eine ausführlichere Syntax, mit der Sie die Typen explizit angeben und mehrere Anweisungen ausführen können. Beispiel:

[code xml:lang="csharp"]return EnumerableExtensions.Where(customers, (Customer c) => { int zip = zipCode; return c.ZipCode == zip; });[/code]

Die gute Nachricht ist, dass wir uns der idealen Syntax schon viel mehr angenähert haben, als im ursprünglichen Bericht dargelegt war, und dies unter Verwendung eines Sprachfeatures, das in der Regel außerhalb von Abfrageoperatoren nützlich ist. Dies ist der momentane Stand:

[code xml:lang="csharp"]IEnumerable locals = EnumerableExtensions.Where(customers, c => c.ZipCode == 91822);[/code]

Hier liegt ein offensichtliches Problem vor. Statt über die Vorgänge nachzudenken, die mit Customer durchgeführt werden können, muss der Empfänger in diesem Stadium über die EnumerableExtensions-Klasse Bescheid wissen. Außerdem muss im Fall mehrerer Operatoren das Denken so umgekehrt werden, dass die richtige Syntax geschrieben wird. Beispiel:

[code xml:lang="csharp"]IEnumerable locals = EnumerableExtensions.Select(EnumerableExtensions.Where(customers, c => c.ZipCode == 91822), c => c.Name);[/code]

Angenommen, Select ist die äußere Methode, obwohl ihre Funktion auf dem Ergebnis der Where-Methode basiert. Die ideale Syntax würde vielmehr folgendermaßen aussehen:

[code xml:lang="csharp"]sequence locals = customers.where(ZipCode == 98112).select(Name);[/code]

Es stellt sich also die Frage, ob es möglich wäre, mit einem anderen Sprachfeature eine größere Ähnlichkeit mit der idealen Syntax zu erzielen.

Erweiterungsmethoden

Es stellte sich heraus, dass eine viel bessere Syntax verfügbar war, und zwar in Form eines Sprachfeatures, das als Erweiterungsmethoden bekannt ist. Erweiterungsmethoden sind im Grunde statische Methoden, die durch eine Instanzsyntax aufgerufen werden können. Das Grundproblem für die obige Abfrage besteht darin, dass IEnumerable Methoden hinzugefügt werden sollen. Wenn jedoch Operatoren hinzugefügt würden, z. B. Where, Select usw., dann müssten alle derzeitigen und künftigen Implementierer diese Methoden implementieren. Die große Mehrzahl der Implementierungen wäre jedoch gleich. Die einzige Möglichkeit, die „Benutzeroberflächenimplementierung“ in C# freizugeben, ist das Verwenden statischer Methoden, was mit der zuvor verwendeten EnumerableExtensions-Klasse erfolgt ist.

Angenommen, die Where-Methode soll stattdessen als eine Erweiterungsmethode programmiert werden. Die Abfrage könnte dann wie folgt umgearbeitet werden:

[code xml:lang="csharp"]IEnumerable locals = customers.Where(c => c.ZipCode == 91822);[/code]

Bei dieser einfachen Abfrage ist die Syntax nun schon sehr nahe am Ideal. Doch was bedeutet es im Einzelnen, die Where-Methode als eine Erweiterungsmethode zu programmieren? Der Vorgang ist eigentlich ziemlich einfach. Im Grunde ändert sich die Signatur der statischen Methodenänderungen so, dass dem ersten Parameter ein „this“-Modifizierer hinzugefügt wird:

[code xml:lang="csharp"]public static IEnumerable Where(this IEnumerable items, Func predicate)[/code]

Hinzu kommt, dass die Methode innerhalb einer statischen Klasse deklariert werden muss. Eine statische Klasse ist eine Klasse, die nur statische Mitglieder enthält und vom statischen Modifizierer in der Klassendeklaration angegeben wird. Das ist auch schon alles. Diese Deklaration weist den Compiler an, den Aufruf von Where mit derselben Syntax wie eine Instanzmethode für einen beliebigen Typ, der IEnumerable implementiert, zu ermöglichen. Die Where-Methode muss jedoch vom aktuellen Bereich zugänglich sein. Eine Methode befindet sich im Bereich, wenn der Typ, der sie enthält, im Bereich ist. Daher ist es möglich, Erweiterungsmethoden durch die Using-Direktive in den Bereich einzubringen. (Weitere Informationen finden Sie in der Randleiste „Erweiterungsmethoden“).

[hidebox]

Es ist klar, dass Erweiterungsmethoden unsere Beispielabfrage vereinfachen, doch sind sie außerhalb dieses Szenarios im Allgemeinen ein nützliches Sprachfeature? Es stellt sich heraus, dass es viele verschiedene Verwendungsmöglichkeiten für Erweiterungsmethoden gibt. Eine der häufigsten ist vermutlich das Bereitstellen freigegebener Benutzeroberflächenimplementierungen. Angenommen, es liegt die folgende Benutzeroberfläche vor:

[code xml:lang="csharp"]interface IDog { // Barks for 2 seconds void Bark(); void Bark(int seconds); }[/code]

Dabei ist erforderlich, dass jeder Implementierer eine Implementierung für beide Überladungen erstellt. Mit der „Orcas“-Version von C# könnte die Benutzeroberfläche einfach so aussehen:

[code xml:lang="csharp"]interface IDog { void Bark(int seconds); }[/code]

Eine Erweiterungsmethode könnte in einer anderen Klasse hinzugefügt werden:

[code xml:lang="csharp"]static class DogExtensions { // Barks for 2 seconds public static void Bark(this IDog dog) { dog.Bark(2); } }[/code]

Jetzt muss der Implementierer der Oberfläche nur eine einzelne Methode implementieren, doch die Clients der Oberfläche können beide Überladungen frei aufrufen.

[/hidebox]

Es steht nun eine Syntax zur Verfügung, die dem Ideal sehr nahe kommt, was die Filterklausel angeht, doch ist das schon alles, was die „Orcas“-Version von C# ausmacht? Nicht ganz. Als Nächstes soll das Beispiel etwas ausgedehnt werden, indem nur der Name des Kunden projiziert wird, nicht das gesamte Kundenobjekt. Wie weiter oben erwähnt, sieht die ideale Syntax wie folgt aus:

[code xml:lang="csharp"]sequence locals = customers.where(ZipCode == 98112).select(Name);[/code]

Unter ausschließlicher Verwendung der bereits erörterten Spracherweiterungen, Lambda-Ausdrücke und Erweiterungsmethoden könnte die Syntax folgendermaßen umgeschrieben werden:

[code xml:lang="csharp"]IEnumerable locals = customers.Where(c => c.ZipCode == 91822).Select(c => c.Name);[/code]

Beachten Sie, dass der Rückgabetyp für diese Abfrage anders ist, und zwar IEnumerable und nicht IEnumerable. Das liegt daran, dass nur der Name des Kunden von der Select-Anweisung zurückgeben werden soll. Dies funktioniert wirklich gut, wenn es sich bei der Projektion nur um ein einzelnes Feld handelt. Angenommen, nicht nur der Name, sondern auch die Adresse des Kunden soll zurückgegeben werden. Die ideale Syntax könnte dann folgendermaßen aussehen:

[code xml:lang="csharp"]locals = customers.where(ZipCode == 98112).select(Name, Address);[/code]

Anonyme Typen

Bei weiterer Verwendung der vorhandenen Syntax zum Zurückgeben des Namens und der Adresse tritt schnell das Problem auf, dass es keinen Typ gibt, der nur einen Namen und eine Adresse enthält. Diese Abfrage könnte jedoch trotzdem durch Einführen des folgenden Typs erstellt werden:

[code xml:lang="csharp"]class CustomerTuple { public string Name; public string Address; public CustomerTuple(string name, string address) { this.Name = name; this.Address = address; } }[/code]

Mit diesem Typ, hier CustomerTuple, kann das Ergebnis der Abfrage erstellt werden:

[code xml:lang="csharp"]IEnumerable locals = customers.Where(c => c.ZipCode == 91822) .Select(c => new CustomerTuple(c.Name, c.Address));[/code]

Das sieht nach jeder Menge Standardcode aus, um lediglich eine Teilmenge von Feldern zu projizieren. Es ist oft auch unklar, wie ein solcher Typ genannt werden soll. Ist CustomerTuple wirklich ein guter Name? Was ist, wenn wir stattdessen Namen und Alter projiziert hätten? Auch

dies könnte ein CustomerTuple sein. Zum einen gibt es das Problem, dass häufig verwendeter Code vorliegt, und zum anderen das Problem, dass es keine guten Namen für die erstellten Typen gibt. Abgesehen davon könnten viele verschiedene Typen erforderlich sein, deren Verwaltung rasch zu einer Qual werden könnte.

Genau dafür sind anonyme Typen gedacht. Dieses Feature ermöglicht grundsätzlich das Erstellen struktureller Typen ohne eine Angabe des Namens. Wenn die Abfrage unter Verwendung anonymer Typen erstellt wird, sieht sie folgendermaßen aus:

[code xml:lang="csharp"]locals = customers.Where(c => c.ZipCode == 91822) .Select(c => new { c.Name, c.Address });[/code]

Dieser Code erstellt implizit einen Typ, der die Felder „Name“ und „Address“ aufweist:

[code xml:lang="csharp"]class { public string Name; public string Address; }[/code]

Auf diesen Typ kann nicht namentlich verwiesen werden, da er keinen Namen hat. Die Namen der Felder können explizit bei der anonymen Typerstellung deklariert werden. Wenn zum Beispiel das erstellte Feld von einem komplizierten Ausdruck abgeleitet ist oder wenn der Name einfach nicht in Frage kommt, ist es möglich, den Namen zu ändern:

[code xml:lang="csharp"]locals = customers.Where(c => c.ZipCode == 91822) .Select(c => new { FullName = c.FirstName + " " + c.LastName, HomeAddress = c.Address });[/code]

In diesem Fall enthält der generierte Typ die Felder „FullName“ und „HomeAddress“. Das bringt uns zwar dem Ideal näher, aber es gibt da ein Problem. Sie werden feststellen, dass ich den Typ der lokalen Variablen überall da, wo ein anonymer Typ verwendet wurde, strategisch ausgelassen habe. Natürlich können die Namen anonymer Typen nicht angeben werden. Wie sollen diese also verwendet werden?

Implizit typisierte lokale Variable

Es gibt ein anderes Sprachfeature, das als implizit typisierte lokale Variable (kurz „var“ genannt) bekannt ist. Dieses Feature weist den Compiler an, den Typ einer lokalen Variablen abzuleiten. Beispiel:

[code xml:lang="csharp"]var integer = 1;[/code]

In diesem Fall hat die Ganzzahl den Typ „int“. Sie sollten unbedingt beachten, dass dies immer noch stark typisiert ist. In einer dynamischen Sprache könnte sich der Ganzzahltyp später noch ändern. Zur Illustration hier ein Code, der nicht kompiliert:

[code xml:lang="csharp"]var integer = 1; integer = "hello";[/code]

Der C#-Compiler meldet einen Fehler in der zweiten Zeile, mit der Angabe, dass er eine Zeichenfolge nicht implizit in „int“ konvertieren kann. Im Fall der obigen Abfrage kann jetzt die vollständige Zuweisung folgendermaßen erstellt werden:

[code xml:lang="csharp"]var locals = customers .Where(c => c.ZipCode == 91822)
.Select(c => new { FullName = c.FirstName + " " + c.LastName,
HomeAddress = c.Address });[/code]

Der Typ der lokalen Variablen ist letzten Endes IEnumerable, wobei „?“ den Namen eines Typs darstellt, der nicht erstellt werden kann (da anonym). Implizit typisierte lokale Variable sind nichts anderes: eine lokale Variable innerhalb einer Methode. Sie können nicht außerhalb der Grenzen einer Methode, Eigenschaft, Indexerstellung oder eines anderen Blocks existieren, da der Typ nicht explizit angegeben werden kann, und „var“ ist für Felder oder Parametertypen nicht zulässig.

Implizit typisierte lokale Variable erweisen sich auch außerhalb des Kontexts einer Abfrage als praktisch. Zum Beispiel werden dadurch komplizierte generische Instanziierungen vereinfacht:

[code xml:lang="csharp"]var customerListLookup = new Dictionary();[/code]

Wir haben es weit gebracht mit unserer Abfrage. Die ideale Syntax ist in greifbare Nähe gerückt, und dies wurde mit allgemeinen Sprachfeatures erzielt.

Interessanterweise haben wir Folgendes festgestellt, nachdem sich der Kreis derjenigen, die mit dieser Syntax arbeiteten, erweiterte: Häufig war es notwendig, eine Projektion zu erlauben, um die Begrenzungen einer Methode zu überschreiten. Wie schon festgestellt, wird dies möglich, wenn ein Objekt so erstellt wird, dass sein Konstruktor von Select aus aufgerufen wird. Was geschieht jedoch, wenn es keinen Konstruktor gibt, der genau die festzulegenden Werte aufweist?

Objektinitialisierer

Für diesen Fall gibt es ein C#-Sprachfeature in der bevorstehenden „Orcas“-Version, das als Objektinitialisierer bekannt ist. Objektinitialisierer ermöglichen im Grunde die Zuweisung mehrerer Eigenschaften oder Felder in einem einzigen Ausdruck. Ein gemeinsames Muster für die Objekterstellung ist beispielsweise dieses:

[code xml:lang="csharp"]Customer customer = new Customer(); customer.Name = "Roger";
customer.Address = "1 Wilco Way";[/code]

In diesem Fall gibt es keinen Konstruktor von Customer, der Namen und Adresse erfordert. Es gibt jedoch zwei Eigenschaften, Name und Address, die festgelegt werden können, sobald eine Instanz erstellt ist. Objektinitialisierer ermöglichen die gleiche Erstellung mit der folgenden Syntax:

[code xml:lang="csharp"]Customer customer = new Customer() { Name = "Roger", Address = "1 Wilco Way" };[/code]

Im CustomerTuple-Beispiel weiter oben wurde die CustomerTuple-Klasse durch Aufrufen des Konstruktors erstellt. Dasselbe Ergebnis kann über Objektinitialisierer erzielt werden:

[code xml:lang="csharp"]var locals = customers .Where(c => c.ZipCode == 91822) .Select(c => new CustomerTuple { Name = c.Name, Address = c.Address });[/code]

Beachten Sie, dass Objektinitialisierer die Auslassung der Konstruktorklammern erlauben. Darüber hinaus können sowohl Felder als auch einstellbare Eigenschaften innerhalb des Texts des Objektinitialisierers zugewiesen werden.

Jetzt steht eine kurze Syntax für das Erstellen von Abfragen in C# zur Verfügung. Darüber hinaus liegt jedoch auch eine erweiterbare Möglichkeit vor, neue Operatoren (Distinct, OrderBy, Sum usw.) durch Erweiterungsmethoden und einen unterschiedlichen Satz von Sprachfeatures hinzuzufügen, die jedes für sich nützlich sind.

Das Sprachentwurfsteam hatte jetzt mehrere Prototypen vorliegen, zu denen Kommentare benötigt wurden. Daher wurde eine Umfrage zur Benutzerfreundlichkeit durchgeführt, bei der viele Teilnehmer Erfahrung sowohl mit C# als auch SQL hatten. Das Feedback war fast durchgehend positiv, doch es wurde deutlich, dass etwas fehlte. Entwickler fanden es insbesondere schwierig, ihr Wissen über SQL anzuwenden, weil die Syntax, die wir als ideal ansahen, sich nicht so gut mit ihrer Domänenfachkenntnis deckte.

Abfrageausdrücke

Das Sprachentwurfsteam hat daraufhin eine Syntax entworfen, die SQL eher entspricht, die Abfrageausdrücke. Zum Beispiel könnte ein Abfrageausdruck für das obige Beispiel folgendermaßen aussehen:

[code xml:lang="csharp"]var locals = from c in customers where c.ZipCode == 91822 select new { FullName = c.FirstName + " " + c.LastName, HomeAddress = c.Address };[/code]

Abfrageausdrücke sind auf den oben beschriebenen Sprachfeatures aufgebaut. Sie sind buchstäblich syntaktisch in die zugrunde liegende Syntax übersetzt, die hier bereits besprochen wurde. Zum Beispiel wird die obige Abfrage direkt so übersetzt:

[code xml:lang="csharp"]var locals = customers .Where(c => c.ZipCode == 91822) .Select(c => new { FullName = c.FirstName + " " + c.LastName, HomeAddress = c.Address });[/code]

Abfrageausdrücke unterstützen eine Anzahl verschiedener „Klauseln“, z. B. Where, Select, OrderBy, GroupBy, Let und Join. Diese Klauseln lassen sich in die gleichwertigen Operatoraufrufe übersetzen, die wiederum über Erweiterungsmethoden implementiert werden. Die enge Beziehung zwischen den Abfrageklauseln und den Erweiterungsmethoden, die die Operatoren implementieren, erleichtert ihre Kombination, falls die Abfragesyntax keine Klausel für einen erforderlichen Operator unterstützt. Beispiel:

[code xml:lang="csharp"]var locals = (from c in customers where c.ZipCode == 91822 select new { FullName = c.FirstName + " " + c.LastName, HomeAddress = c.Address}) .Count();[/code]

In diesem Fall gibt die Abfrage jetzt die Anzahl der Kunden zurück, die in der Region mit der Postleitzahl 91822 leben.

So kommen wir fast wieder zum Ausgangspunkt zurück (was ich immer sehr zufriedenstellend finde). Die Syntax der nächsten Version von C# hat sich über die letzten Jahre durch mehrere neue Sprachfeatures entwickelt, um letztendlich sehr nahe an die ursprüngliche Syntax heranzukommen, die im Winter 2004 vorgeschlagen wurde. Das Hinzufügen von Abfrageausdrücken basiert auf der Grundlage, die von den anderen Sprachfeatures in der bevorstehenden C#-Version bereitgestellt wird, und trägt dazu bei, dass viele Abfrageszenarios besser lesbar und für Entwickler mit einem Hintergrund in SQL leichter verständlich sind.