TPL, LINQ, RX UND ASYNCHRONE ITERATOREN

C# in Raum und Zeit

Foto: Shutterstock / Lidiia

Mit den richtigen Typen und Compiler-Power Daten in zwei Dimensionen meistern.

An innovativen Neuerungen mangelte es C# bisher nicht in seiner gut 20-jährigen Geschichte. Viele der Sprachmerkmale sollen für mehr Eleganz sorgen, sie weniger „chatty" machen, wie man auch sagt. Entwickler*innen sollten ihre Vorhaben mit weniger Code zum Ausdruck bringen können, als das mit früheren Versionen der Sprache möglich war. Entsprechend wurde die Standard-Klassenbibliothek nach und nach erweitert, sofern es zum Umsetzen einer neuen Sprachsyntax notwendig war. Einige der Sprachfeatures und Erweiterungen stehen, obwohl zu verschiedenen Zeitpunkten eingeführt, in einer interessanten Beziehung zueinander; sie ergänzen sich gegenseitig und erlauben dadurch einen ganz anderen Blick auf Datenverarbeitung als gewohnt. Diese Zusammenhänge aufzuzeigen und vielleicht neue Sichtweisen zu eröffnen, ist das Ziel dieses Artikels.

Listing 1: IEnumerable und IEnumerator

```
public interface IEnumerable<T>
{
  IEnumerator<T> GetEnumerator();
}

public interface IEnumerator<T>
{
  bool MoveNext();
  T Current { get; }
}
```

Features? Welche Features?

Die Sprachmerkmale, um die es hier geht, sind jene, die Entwickler in die Lage versetzen, die Menge der zu verarbeitenden Daten (Werte oder Objektreferenzen) aussagekräftig zu repräsentieren und elegant mit ihr umzugehen, selbst wenn sie sich beliebig in den Dimensionen Raum und Zeit ausdehnt.

Ausdehnen im Raum heißt, nicht nur ein einzelnes Objekt zu betrachten, sondern beliebig viele, sogar unbegrenzt viele – oder auch gar keins. Objekte nehmen Speicherplatz ein, eine Art „Space", also „Raum". Dagegen bedeutet „ausdehnen" im Zeitkontext, dass ein Objekt nicht jetzt vorliegen muss, sondern möglicherweise erst später – oder auch nie.

Diese Ausdehnung in zwei Dimensionen erlaubt es nun im Prinzip, die bekannten Datentypen mit Begriffen wie „ein", „vielleicht ein", „kein", „viele", „jetzt", „später" und „nie" zu quantifizieren und zu ordnen.

Zunächst gilt es, die Dimensionen Raum und Zeit einzeln zu betrachten. Dazu werden die Begriffe in dem Koordinatensystem von Bild 1 durch die ihnen entsprechenden .NET-Typen auf den Dimensionsachsen verortet und das jeweils entsprechende C#-Feature entgegengestellt. Später werden Raum und Zeit dann miteinander kombiniert, sodass sich das

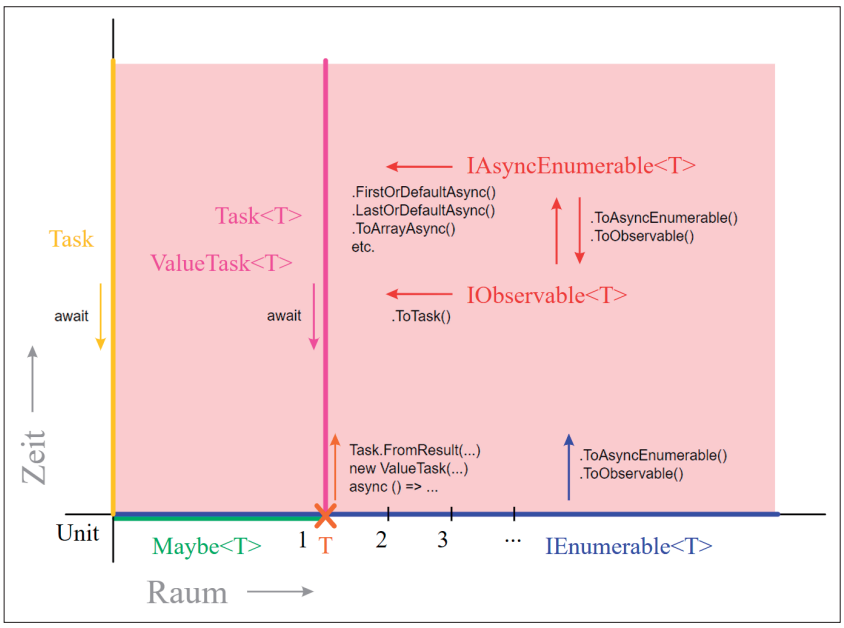
gezeigte Diagramm ergibt. Für alle Beispiele soll immer der grundlegende Datentyp *string* verwendet werden.

Raum

Die isolierte Betrachtung der Raumdimensionsachse bewegt sich, anders als die Zeitdimensionsachse, praktisch nicht über C#-2.0-Merkmale hinaus. Haben Sie bitte trotzdem etwas Geduld während der folgenden Rekapitulation, es lohnt sich.

Die einfachste Möglichkeit, einen Datentyp zu quantifizieren, ist, ihn einfach so zu lassen, wie er ist. Der Typ *string* wird dann unter Hinzunahme eines der Begriffe „Raum" oder „Zeit" zu „ein *string*", und der ist in C# schnell definiert:

```
var einString = "Hallo";
```



Die Raum-Zeit-Ebene für .NET-Datentypen (Bild 1)

Das ist weniger ein Feature von C# als eher ein grundlegendes Merkmal jeder halbwegs benutzbaren Programmiersprache: die Zuweisung eines Werts eines bestimmten Datentyps zu einer Variable. Streng genommen ist die Variable nicht einmal von Belang – die Zeichenkette auf der rechten Seite steht schon für sich alleine.

Die Ausdehnung nach rechts auf der Dimensionsachse scheint nahezuliegen: „Viele *string*-Objekte" – oder generell eine Liste (Array) von Referenzen oder Werten – sind ebenfalls schnell beschrieben:

```
var vieleStrings = new[] { "Hallo", "Auf Wiedersehen" };
```

Die Repräsentationen von „ein *string*" oder „kein *string*" stellen sich nun als Sonderfälle hiervon dar:

```
var einString = new [] { "Hallo" };
var keinString = new string[0];
```

Lässt sich hiermit nun die gesamte Raumdimensionsachse belegen? Für praktische Anwendungen ja, zumindest annähernd: Die Größe eines Arrays in C# ist nach oben nur durch den maximalen Wert eines 32-Bit-Integer-Werts beschränkt (immerhin etwas über 2 Milliarden) und natürlich durch den verfügbaren Speicherplatz. An dieser Stelle soll die erste Abstraktion ins Spiel gebracht werden: Statt ganze Listen von Objekten zu betrachten, die zum selben Zeitpunkt komplett materialisiert im Speicher liegen, beschränkt sich die Abstraktion auf Aufzählungen, die in .NET durch die Schnittstellen *IEnumerable<T>* und *IEnumerator<T>* repräsentiert werden. Die nicht generischen Versionen dieser Schnittstellen waren schon Teil des ersten .NET Framework, die logische Weiterentwicklung mit den sogenannten Generics kam dann 2005 mit .NET 2.0 hinzu.

So grundlegend und bekannt diese Schnittstellen auch sein mögen, sie kurz zu erläutern ist trotzdem sinnvoll, da so später der Zusammenhang zum asynchronen Gegenstück klarer ersichtlich wird. Listing 1 zeigt dazu (in vereinfachter Form) ihre Signaturen.

Konsumenten eines *IEnumerable<T>*-Objekts können mit *GetEnumerator()* eine neue Aufzählung vom Typ *IEnumerator<string>* verlangen. Jeder Aufruf von *MoveNext()* versucht nun, ein Element in der Aufzählung weiterzugehen. Gibt es ein nächstes Element, gibt der Aufruf *true* zurück und das aktuelle Element kann von der *Current*-Eigenschaft abgerufen werden. Falls nicht, so ist die Aufzählung beendet. C# bietet mit dem *foreach*-Schleifenkonstrukt eine bequeme Syntax, die genau diesen Ablauf unter der Haube bewerkstelligt.

Für die produzierende Seite wurde mit C# 2.0 das Iterator-Feature mit dem damals neuen *yield*-Statement einge- ▶

Listing 2: Synchrone Iteratoren

```
static void Main()
{
  foreach (var zahl in UngeradeZahlen())
  {
    Console.WriteLine(zahl);
  }
}

private static IEnumerable<int> UngeradeZahlen()
{
  for (var i = 1; i < int.MaxValue; i += 2)
  {
    yield return i;
  }
}
```

führt. Mit ihm lässt sich Aufzählungen erzeugender Code schreiben, ohne die nötigen Schnittstellen händisch implementieren zu müssen. Ein Beispielprogramm, das alle ungeraden Zahlen aufzählt und ausgibt, ist in Listing 2 enthalten.

Damit ist nun die Repräsentation beliebig langer (und unbegrenzter) Elementfolgen möglich. Ist damit aber nicht schon die Eindimensionalität verlassen? Sind wir damit nicht schon in der Zeitdimension gelandet? Denn immerhin kostet ein wiederholtes Durchlaufen einer *foreach*-Schleife Zeit, im Falle unendlicher Sequenzen sogar unendlich viel davon. Trotzdem erstreckt sich, laut Bild 1, ein *IEnumerable<T>*-Objekt zwar potenziell über die gesamte Raumachse, jedoch ausschließlich auf Höhe des Nullpunkts der Zeitachse.

Der Einwand ist einerseits korrekt, würdigt andererseits aber nicht den Begriff von Zeit, den dieser Artikel vermitteln will. Denn bei dieser Sichtweise liegt streng genommen nicht einmal mehr die einfache Zuweisung einer konstanten Zeichenkette zu einer Variablen (wie oben gesehen) auf der Achse – nichts ist zeitlos, alles dauert mindestens die ein oder andere Nanosekunde; und sollte das Betriebssystem zwecks Multithreading oder Garbage-Collection den ausführenden Thread einmal schlafen legen, so ist man schon im Bereich von Millisekunden.

Stattdessen soll für die weiteren Betrachtungen gelten, dass jeder Vorgang, der synchron abläuft, mit der Zeitdauer Null zu bewerten ist. Das soll selbst dann gelten, wenn der ausführende Thread unterbrochen wird. Entsprechend ist der Signatur von *IEnumerator<T>* schon anzusehen, dass hier alles synchron abläuft – die Einordnung auf die Raumachse auf Höhe der Null-Zeit ist also korrekt.

Wie schon weiter oben erwähnt, befindet sich die leere Aufzählung (oder ein leeres Array) auf dem Ursprung des Koordinatensystems. Gibt es aber noch andere Wege, den Sachverhalt „kein *string*“ zu repräsentieren, beispielsweise die Semantik? Eine Methode in C#, die keinen Wert zurückgibt, zeigt dies durch das Schlüsselwort *void* an – nur leider ist *void* kein Datentyp in C#.

Diese Design-Entscheidung ist der Herkunft von C# geschuldet, denn mathematisch bildet eine Funktion immer aus einer Menge in eine andere Menge ab, selbst wenn die Zielmenge praktisch bedeutungslos ist. Einige Bibliotheken (wie zum Beispiel LanguageExt.Core [1] oder Reactive Extensions [2][3] bieten daher einen sogenannten Unit-Typ an, einen Werttyp ohne Daten mit einer Singleton-Instanz und derselben Semantik wie *void*. Aus diesem Grund ist der Nullpunkt in Bild 1 mit *Unit* beschriftet.

Listing 3: FirstOrNothing()

```
public class Programm
{
  static void Main()
  {
    var a1 = new string[] {null};
    var a2 = new string[0];

    Console.WriteLine($"a1.FirstOrDefault =
      {a1.FirstOrDefault() ?? "null"}");
    Console.WriteLine($"a2.FirstOrDefault =
      {a2.FirstOrDefault() ?? "null"}");
    Console.WriteLine($"a1.FirstOrNothing().
      HasValue = {a1.FirstOrNothing().HasValue}");
    Console.WriteLine($"a2.FirstOrNothing().
      HasValue = {a2.FirstOrNothing().HasValue}");
    Console.ReadLine();
  }
}

static class Extensions
{
  public static Maybe<T> FirstOrNothing<T>(
    this IEnumerable<T> source)
  {
    return source.Select(x =>
      new Maybe<T>(x)).FirstOrDefault();
  }
}
```

Listing 4: Der Maybe-Typ

```
public struct Maybe<T>
{
  public static readonly Maybe<T> Nothing =
    default;

  private readonly T value;
  private readonly bool hasValue;

  public Maybe(T value)
  {
    this.value = value;
    this.hasValue = true;
  }

  public T Value
  {
    get
    {
      if (!hasValue)
        throw new InvalidOperationException(
          "Kein Wert!");

      return value;
    }
  }

  public bool HasValue { get => hasValue; }
}
```

Nullables und Maybes

Die gesamte Raumachse ist also durch den Datentyp *IEnumerable<T>* repräsentierbar, Unit- und skalare Werte sind dabei nur Sonderfälle. Man könnte es nun dabei belassen, wäre da nicht noch ein weiterer Sonderfall dieser Dimension, dessen Betrachtung sich lohnt: ein Datentyp, der die Spanne zwischen null und eins abdeckt; der also repräsentiert, dass entweder kein Element oder genau ein Element vorhanden ist.

Für Referenztypen ist die Situation scheinbar klar – jede *null*-Referenz (in einer *string*-Variable) bildet die Semantik „kein *string*" ab. Für Werttypen, die faktisch immer einen Wert haben (und sei es nur der Default-Wert, zum Beispiel 0 bei Zahlen) hat .NET 2.0 den Typ *Nullable<T>* eingeführt, der dank entsprechender Sprachunterstützung auch für Werttypen eine semantische Unterscheidung zwischen „kein Wert" und „ein Wert" zulässt.

Leider verbieten die generischen Einschränkungen auf *Nullable<T>* aber, wie schon erwähnt, die Verwendung mit Referenztypen. Und leider ist die Semantik von *null* für Referenztypen auch nicht konsequent „kein Wert" – die Information, die eine *null*-Referenz liefert, reicht oft schlicht nicht aus, wie das Beispiel in Listing 3 zeigt: Hier wird der LINQ-Operator *FirstOrDefault()* von einem *string*-Array mit nur einem Element (allerdings *null*) und bei seinem leeren Array aufgerufen. Beide Male gibt die Funktion *null* zurück, was keinen Aufschluss darüber gibt, ob die Sequenz leer oder das erste Element gerade *null* war. Die wesentlich elegantere Erweiterungsmethode *FirstOrNothing()* tut das Gleiche mit einem sogenannten „Maybe-Typ" (Listing 4). Er vereint diese zwei Informationen miteinander, funktioniert mit Wert- und Referenztypen gleichermaßen und eignet sich vor allem als Rückgabetyp einer Methode ohne *out*-Parameter. Somit platziert sich *Maybe<T>* sinnvoll zwischen 0 und 1 auf der Raumachse.

Zeit

Nachdem die Raumachse nun durch verschiedene Datentypen erschöpfend belegt ist, schwenkt der Blick auf die Zeitachse in vertikaler Ausrichtung. Im Nullpunkt bleibt der *Unit*-Typ (*void*) bestehen: kein Wert nach keiner Zeit, also „kein Wert, jetzt". Anders als die Raumachse ist die Zeitachse nicht in diskrete Abschnitte unterteilt, sondern stellt ein Kontinuum dar, wie die echte Zeit eben auch. Das gesamte Kontinuum im Zeitbereich wird für skalare Werte vom Typ *Task<T>* abgedeckt. Für den Fall, dass das Ergebnis nicht von Belang ist, kann auch der Basistyp *Task* verwendet werden. *Task<T>* hat die Semantik „ein Wert vom Typ *T*, irgendwann, möglicherweise auch nie".

Die *Task*-Typen sind Teil der Task Parallel Library, die seit .NET 4 Teil des Frameworks ist und sich damit auch in .NET Standard/Core/5 wiederfindet. Mit .NET Core 2.0 (und für alles davor als externes NuGet-Paket) wurde zusätzlich der Werttyp *ValueTask<T>* eingeführt, der grundsätzlich dieselbe Semantik hat wie *Task<T>* – für den Fall allerdings, dass ein Wert nicht erst in der Zukunft erwartet wird, sondern schon verfügbar ist und vollständig ohne Allokationen auf dem Heap auskommt. Wichtig: *ValueTask<T>* bedeutet nicht, dass *T* ein Werttyp sein muss, sondern nur, dass *ValueTask<T>* selbst ein Werttyp ist.

So weit, so vermutlich bekannt. Noch gar nicht so alt (und bis vor Kurzem relativ obskur) war die Repräsentation der gesamten Ebene in Bild 1, also beliebig viele Werte beliebig in der Zeit verteilt. Wie ist das zu erreichen? Beliebig viele Werte wurden bislang für den synchronen Fall durch den Typ *IEnumerable<T>* repräsentiert. Es lohnt sich also noch mal ein Blick in Listing 1, speziell auf die Schnittstelle *IEnumerator<T>*. Mit Blick auf die am Anfang des Artikels getroffene Konvention, dass synchrone Vorgänge praktisch als zeitlos zu betrachten sind, muss jetzt also die Zeit in die Signatur einfließen. Da kommt nur die *MoveNext()*-Methode infrage, die – siehe oben – die Aufzählung um ein Element weiterschiebt (und *true* zurückgibt) oder das Ende der Sequenz anzeigt (und damit *false* zurückgibt). Aus *MoveNext()* wird also *MoveNextAsync()* und aus dem Rückgabetyp *bool* der Rückgabetyp *ValueTask<bool>*; fertig sind *IAsyncEnumerable<T>* und *IAsyncEnumerator<T>*, bis auf ein paar fehlende Details, die für diesen Artikel nicht weiter wichtig sind. Listing 5 zeigt diese Schnittstellen; in dieser Ausprägung sind sie Teil von .NET Standard 2.1 und des Support-Pakets Async.Bcl.Interface (für Plattformen, die .NET Standard 2.1 nicht implementieren).

Mit C# 8 hat Microsoft Parität hergestellt zwischen synchronen und asynchronen Aufzählungen. Sowohl beim Erzeugen als auch beim Konsum (mit der neuen Syntax *await foreach*) ist *IAsyncEnumerable<T>* ebenso anzuwenden wie *IEnumerable<T>*. Gleichstand herrscht auch bei LINQ, dessen Erweiterungen und Operatoren in den Paketen System.Linq.Async [4] und System.Interactive.Async [5] zu Hause sind. Listing 6 zeigt eine kleine Konsolenanwendung, die von einer asynchronen Aufzählung jede Sekunde die aktuelle Uhrzeit geliefert bekommt, diese mittels LINQ auf *string*-Objekte projiziert und dann auf der Konsole ausgibt. Der komplette Code ist auch Teil des Beispielprojekts, das Sie auf der Homepage der dotnetpro herunterladen können. Damit stehen nun mächtige Werkzeuge zur Verfügung, um Erzeugung und Verarbeitung von diskreten Daten, die über die Zeit verteilt anfallen, sauber und elegant zu repräsentieren und zu verarbeiten. ▶

Listing 5: IAsyncEnumerable und IAsyncEnumerator

```
public interface IAsyncEnumerable<T>
{
  IAsyncEnumerator<T> GetAsyncEnumerator(
    CancellationToken ct);
}

public interface IAsyncEnumerator<T>
{
  ValueTask<bool> MoveNextAsync();
  T Current { get; }
}
```

Eine dritte Dimension

Doch das ist nicht die ganze Wahrheit. Tatsächlich hat der vorherige Abschnitt zur Zeitachse ein paar Vereinfachungen angenommen, was das Wesen der *Task*-Typen angeht. Diese stellen nämlich nicht nur die Ausführung einer Aufgabe dar, die auf jeden Fall mit einem Ergebnis daherkommt (sofern sie denn in endlicher Zeit beendet wird), sondern können auch noch zwei andere Ausgänge zur Folge haben: Die Aufgabe kann zu einem Fehler führen, dann ist das *Task*-Objekt im *IsFaulted*-Zustand und jeder Versuch danach, auf das Ergebnis zuzugreifen (entweder mit *await* oder durch Abrufen der *Result*-Eigenschaft), wird unweigerlich eine Ausnahme erzeugen. Darüber hinaus lässt sich eine Berechnung auch noch abbrechen (über den *Cancelled*-Zustand), was im Prinzip zum selben Verhalten führt, nur mit geänderter Exception. Konzeptuell ließe sich nun also noch eine dritte diskrete Dimension für den Fehlerfall beschreiben, wobei sich diese nur auf „Erfolg" und „Fehler" beschränken würde. Die sich ergebenden möglichen Semantiken („ein Wert oder ein Fehler", „ein Wert oder ein Fehler irgendwann in der Zukunft") sind interessant und nützlich, sprengen aber den Rahmen dieses Artikels; die entsprechenden Typen werden vom schon mehrfach erwähnten Paket LanguageExt.Core [1] auch teilweise implementiert.

Push und Pull

Bild 1 listet neben *IAsyncEnumerable<T>* auch noch *IObservable<T>* auf. Für den Blickwinkel, den dieser Artikel einnimmt, haben die beiden Typen dieselbe Semantik. Während *IAsyncEnumerable<T>* aber eine Aufzählung repräsentiert, bei der jedes nächste Element aktiv vom Konsumenten angefordert wird (Pull-Modell), stellt *IObservable<T>* eine Quelle von Ereignissen dar, die den Beobachter bei jedem neuen Element (oder bei einem Fehler oder dem Ende der Sequenz) durch einen Rückruf benachrichtigt. Das Modell wird damit zum Push-Modell (aus Sicht des Erzeugers) und der Konsument handelt reaktiv. Die gebräuchlichste Implementierung dieses Musters für .NET/Reactive Extensions [2][3] war schon Gegenstand einiger Artikel, sowohl für die .NET- [6] als auch für die JavaScript-Variante [7][8]), und soll daher an dieser Stelle nicht mehr vorgestellt werden.

Hin und her und kreuz und quer

Nun stellt sich die Frage: Wenn jetzt etwa *IAsyncEnumerable<string>* oder *Task<string>* oder auch nur ein diskreter *string* zur Hand ist – wie lässt sich an den Dimensionen schrauben, wie zwischen den Repräsentationen wechseln?

Grundsätzlich wurde ja schon deutlich, dass alles, was sich näher zum Koordinatenursprung in Bild 1 hin befindet, ein Sonderfall ist und die Typen abstrakter werden, je weiter man vom Ursprung weggeht. „Ein" und „kein" sind also nur Sonderfälle von „viele", „jetzt" ist nur ein Sonderfall von „irgendwann" und so weiter. Die trivialste Umwandlung ist die zwischen keinem/einem und vielen Werten, sie wurde oben schon gezeigt. Auch aus einer Variable *str* einen *Task<string>* oder *ValueTask<string>* zumachen, ist mit *Task.FromResult(str)*

Listing 6: Asynchrone Iteratoren

```
static async Task Main(string[] args)
{
  var dateTimeStrings = GetTime()
    .Select(x => x.ToString("G"));

  await foreach (var str in dateTimeStrings)
  {
    Console.Write($"\r{str}");
  }
}

private static async IAsyncEnumerable<DateTime>
    GetTime([EnumeratorCancellation]
    CancellationToken ct = default)
{
  while (true)
  {
    yield return DateTime.Now;
    await Task.Delay(TimeSpan.FromSeconds(1),
      ct);
  }
}
```

Listing 7: Parallelisierung

```
static async Task Main()
{
  var sekunden = new[] { 2, 3, 5, 8, 13, 21 };

  var parallel = sekunden
    .ToObservable()
    .SelectMany(async (i, ct) =>
    {
      Console.WriteLine($"Aufgabe gestartet: "
        + $"Warte {i} Sekunden.");
      await Task.Delay(
        TimeSpan.FromSeconds(i), ct);

      return DateTime.Now;
    })
    .ToAsyncEnumerable();

  await foreach (var time in parallel)
  {
    Console.WriteLine(
      $"Aufgabe beendet um {time:G}.");
  }
}
```

oder *new ValueTask<string>(str)* schnell erledigt; bei Funktionen mit *async*-Signatur hilft der Compiler.

Auch eine synchrone *IEnumerable<string>*-Auflistung ist nur ein Sonderfall von *IAsyncEnumerable<string>*, nämlich der, der jedes Element sofort parat hat und nicht verzögert; zur Umwandlung in die Zeitdimension stellt das Paket System.Linq.Async [4] die Methode *ToAsyncEnumerable()* bereit. Ähnlich verhält es sich mit der Beziehung von *Task<string>* zu *IAsyncEnumerable<string>*, denn *Task<string>* basiert auf *IAsyncEnumerable<string>*, das nach einem Wert terminiert.

Damit sind wir irgendwo in der Ebene aus Bild 1 gelandet. Führt auch ein Weg wieder zurück in Richtung einer der Achsen? Angenommen, es liegt eine *IAsyncEnumerable<string>* vor, also viele in der Zeit verteilte Zeichenketten: Sich nach links Richtung Zeitachse zu bewegen, heißt, den Raum zu verkleinern – also aus vielen wenige Werte machen.

Je nachdem, wozu die ursprünglichen Daten benötigt werden, klappt das auf verschiedene Weisen: Man könnte die Werte mit *ToArrayAsync()* über die Zeit hinweg sammeln und erhielte so einen *Task<string[]>*. Aus „viele *strings* über die Zeit hinweg" würde also „ein Array von *strings*, irgendwann". Hier ist es von Vorteil, wenn die Ursprungssequenz endlich ist, da *Task<string[]>* sonst bis in alle Ewigkeit Werte sammelt. Sofern nicht alle Elemente aus der Sequenz von Interesse sind, sondern nur ein Teil oder eine Aggregation, lässt sich mit den bekannten LINQ-Operatoren (*Take*, *Range*, *Skip*, *FirstAsync*, *Aggregate* etc.) der Raum verkleinern.

Was würde es hingegen bedeuten, sich Richtung Raumachse zu bewegen, also aus einem „irgendwann" ein „jetzt" zu machen? Natürlich kann aus „irgendwann" nur „jetzt" werden, allein durch Abwarten. Asynchrones Warten (Warten, ohne dafür einen Thread zu blockieren) bedeutet seit C# 5: *await*. *Await* ist damit der Projektionsoperator, der jeden *Task* oder *ValueTask* auf die Raumachse bringt. Mit *IAsyncEnumerable<T>* lässt sich das ohne Weiteres nicht erreichen, hier muss, wie schon gezeigt, erst nach links auf einen passenden *(Value)Task*-Typ projiziert werden, der dann mit *await* ins Jetzt geholt wird. Wichtig: Bitte kommen Sie nicht auf die – vielleicht naheliegende – Idee, auf einer asynchronen Sequenz einfach *ToEnumerable()* auszuführen (diese Methode gibt es nämlich auch). Jeder Aufruf von *MoveNext()* würde den Thread blocken, und das ist in modernem Code natürlich unter allen Umständen zu vermeiden.

Zuletzt sei doch noch einmal auf *IObservable<T>* eingegangen, denn auch diese Repräsentation einer asynchronen Sequenz lässt sich nahezu beliebig auf eine der Achsen projizieren. Auch eine Umwandlung von und hin zu *IAsyncEnumerable<T>* ist möglich, allerdings ist es wichtig, zu verstehen, was diese Umwandlung bedeutet. Eine bestehendes *IObservable<T>*-Objekt mit *ToAsyncEnumerable()* in eine asynchrone Sequenz zu überführen heißt, dass die Elemente von *IObservable* empfangen und eventuell zwischengespeichert werden müssen, sobald der Konsument mit der Aufzählung beginnt. Hier ist Vorsicht geboten vor einem Ungleichgewicht in den Geschwindigkeiten von Produzent und Konsument (Stichwort Backpressure/Rückstau). Den umgekehrten Weg geht *ToObservable()* von *IAsyncEnumerables*: Sofern das resultierende *IObservable<T>*-Objekt beobachtet wird, werden quasi in Dauerschleife Elemente aus der ursprünglichen Sequenz angefordert, mit denen der Beobachter dann „beworfen" wird. Alle gerade genannten Transformationen sind auch in Bild 1 eingezeichnet.

Ein sehr spannender Anwendungsfall für einige der genannten Transformationen ist in Listing 7 zu sehen: Ein Array, das einige Wartezeiten in Sekunden enthält, wird in ein Observable konvertiert, das, sobald man es „anschmeißt", alle Werte aus dem Array produziert. Daraufhin wird jede dieser Sekundenangaben auf ein *Task<DateTime>*-Objekt projiziert, das entsprechend lange verzögert und daraufhin die aktuelle Uhrzeit zurückgibt. Am Ende dieser Pipeline wird das Ganze zurück in *IAsyncEnumerable<DateTime>* umgewandelt, das mit *await foreach* durchlaufen wird. An der Ausgabe zeigt sich, dass hier auf einfache Art und Weise alle asynchronen Aufgaben parallelisiert wurden – ganz ohne manuelles Erzeugen von Tasks oder gar Threads.

Fazit

Mit Daten, die nicht nur im Moment vorliegen, sondern in der Zeit verteilt sind, lässt sich mit der richtigen Repräsentation, der Mächtigkeit von C# und unterstützenden NuGet-Bibliotheken leicht jonglieren. Seit bei C# 8 asynchrone Iteratoren dazugekommen sind, ist die Raum-Zeit-Ebene komplett abgedeckt und es ist möglich, praktisch jedes Szenario mit vorher unerreichter Ausdruckskraft in Code abzubilden. ■

[1] language-ext, C# Functional Programming Language Extensions, www.dotnetpro.de/SL2104TPLZeit1

[2] Reactive Extensions, www.dotnetpro.de/SL2104TPLZeit2

[3] NuGet Gallery, Reactive Extensions (Rx) for .NET, www.dotnetpro.de/SL2104TPLZeit3

[4] NuGet Gallery, System.Linq.Async, www.dotnetpro.de/SL2104TPLZeit4

[5] NuGet Gallery, System.Interactive.Async, www.dotnetpro.de/SL2104TPLZeit5

[6] Jannik Lassahn, Heiko Lewandowski, Lass es fließen, Rx.NET, dotnetpro 3/2018, Seite 58 ff., wwww.dotnetpro.de/A1803ReactiveNET

[7] Thorsten Hans, Auf zu reaktivem Code, ReactiveX – Teil 1, dotnetpro 7/2017, Seite 85 ff., www.dotnetpro.de/A1707ReactiveX

[8] Thorsten Hans, Bunter Strauß Operatoren, ReactiveX – Teil 2, dotnetpro 8/2017, Seite 95 ff., www.dotnetpro.de/A1708ReactiveX

Daniel Weber
studierte Informatik und entwickelt seit 2002 für die .NET Plattform. Seine Schwerpunkte liegen u. a. im Bereich Graphdatenbanken und asynchrone/reaktive Programmierung unter .NET. Er ist zu erreichen unter @danielcweber oder **me@danielcweber.net**.

dnpCode A2105TPLZeit