Expression trees

Espresso voor je code!

Jan Jongboom

De in het .Net framework 3.5 geïntroduceerde 'expression trees', zijn absoluut het Microsoft equivalent van DeLonghi's famous espresso: goed gedoseerd levert het een ongelooflijke boost, maar een kopje teveel, en je stuitert alle kanten op.

In dit artikel gaan we in op de vraag wat expression trees zijn en hoe ze delen van je code tot honderd keer sneller kunnen maken. Laten we bij het begin beginnen: wat zijn expressions eigenlijk? Wellicht het meest eenvoudige stuk code demonstreert dit het beste:

1 + 2;

Dit is een expression pur sang. Het is namelijk: 'An instruction to execute something that will return a value.' Uiteraard kunnen we deze expression ook wat moeilijker maken, door twee variabelen te gebruiken:

a + b;

Omdat we soms de expression niet direct willen uitvoeren, kunnen we deze expression ook in een delegate stoppen. Een delegate is namelijk niets anders dan een expression, een uitvoerbaar stuk code, in een variabele:

```
PlusDelegate plus = delegate(int a, int b) { return a + b; };
```

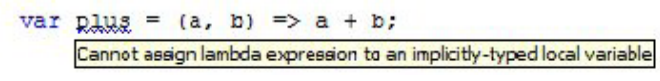
Dit stuk code kunnen we doorgeven naar andere delen van ons programma, en uitvoeren wanneer we willen. In .Net 3.5 kunnen we dit zelfs nog eleganter schrijven als 'lambda expression':

```
PlusDelegate plus = (a, b) => a + b;
```

Of zelfs zonder vooraf aangemaakte delegate:

```
Func<int, int, int> plus = (a, b) => a + b;
```

Wie echter wel eens heeft geprobeerd het nieuwe 'var' keyword te gebruiken bij een lambda expressie komt echter bedrogen uit:



Een lambda is namelijk helemaal geen delegate. Een lambda kan alle vormen aannemen die het wil. En hier is een massive breakdown van het standaard concept in het .Net framework. Niet de expressie bepaalt wat het type moet worden (1+1 geeft een integer terug), maar het type waaraan de expressie wordt toegewezen bepaalt wat de expressie teruggeeft.Een lambda kan namelijk ook heel iets anders dan een delegate worden; een expression tree:

```
Expression<Func<int,int,int>> plus = (a, b) => a + b;
```

Een expression tree is, anders dan een delegate, geen uitvoerbaar stuk code; maar een 'representatie' van het uitvoerbare stuk code. We delen onze 1 + 2 expressie op in delen:

1. Het linkerdeel van de expressie. Het constante getal '1'.
2. De 'plus' operator.
3. Het rechterdeel van de expressie. Het constante getal '2'.

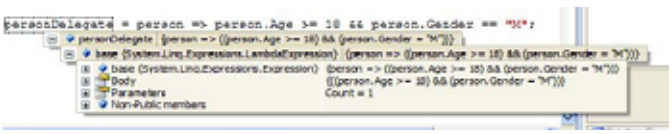
Dit is wat je terugkrijgt van het .Net framework:
Een object van het type 'BinaryExpression': een expressie met exact wat we hierboven hebben gezegd. Een linkerstuk, een rechterstuk, en de operator. Om het concept verder te verduidelijken: een voorbeeld stuk code wat gebruikt kan worden in LINQ to SQL (een manier waarop je via expressions kunt praten met SQL Server).

```
Expression<Func<Person, bool>> personDelegate = person => person.
Age >= 18 && person.Gender == 'M';
```

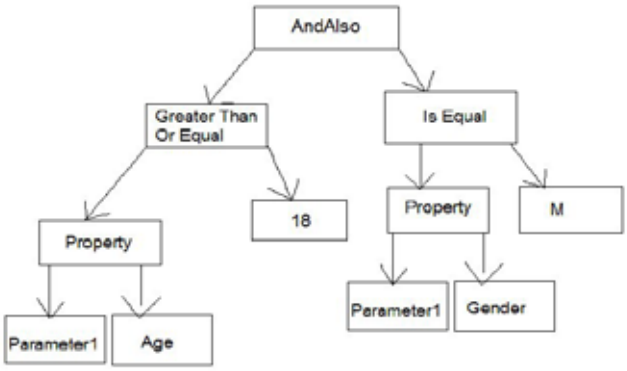
Als we kijken naar het relevante stuk, vanaf 'person =>', dan kunnen we de expressie hierboven opnieuw opdelen: We hebben twee expressies: de vergelijking op leeftijd (linkerexpressie), en de vergelijking op geslacht (rechterexpressie).

1. De linkerexpressie is op te delen in twee expressies:
   a. De verwijzing naar 'person.Age'
   b. De constante waarde '18'
2. De rechterexpressie idem dito:
   a. De verwijzing naar 'person.Gender'
   b. De constante waarde 'M'

Als we de code uitvoeren, krijgen we in plaats van een delegate, netjes een boom terug met alle condities. Dit testen is heel eenvoudig, plak de code in Visual Studio, en voer het programma uit:



Zoals te zien is, krijgen we nu een object terug van het type 'LambdaExpression', met een eigenschap 'body' (met daarin het stuk wat we bedacht hebben), en 1 parameter (de 'person' die we meegeven). Wanneer je 'Body' uitklapt, zal je zien dat deze ook weer een expressie is: namelijk een 'BinaryExpression' van het type 'AndAlso'; die zelf weer een linker en een rechterexpressie heeft, zoals we eerder zeiden in het voorbeeld. Je kunt zo ver gaan in de boom als je zelf wilt, maar alle losse stappen zijn dus uitgetekend door de compiler.
Als we de boom helemaal uittekenen, krijgen we een figuur als onderstaand:



Als we deze boom kunnen uitlezen in code, dan kunnen we er eenvoudig een SQL query van maken:

```
WHERE Age >= 18 AND Gender = 'M'
```

Laten we de SQL query eens opbouwen vanuit de expressie tree.

```
string sql = "WHERE ";
```

Eerst moeten we de tree pakken:

```
LambdaExpression lambda = (LambdaExpression)personDelegate;
```

Zoals je kunt zien als je je code aan het uitvoeren bent, heeft deze variabele een expressie onder zich: 'Body', die van het type 'BinaryExpression' was.

```
BinaryExpression binary = (BinaryExpression)lambda.Body;
```

Een BinaryExpression heeft twee expressies in zich (zie de boom): de 'Greater than or equal', en de 'Is equal'. Beiden zijn weer BinaryExpressions. De linkerexpressie ('Greater than or equal') bestaat uit een property en een constante waarde:

```
BinaryExpression left = (BinaryExpression)binary.Left;

MemberExpression property1 = (MemberExpression)left.Left;

ConstantExpression constant1 = (ConstantExpression)left.Right;
Nu willen we de naam van 'property1' toevoegen aan de sql expres-
sie:
sql += property1.Member.Name;
Hierna willen we de GreaterThanOrEqual toevoegen. De volgende
 helper functie wordt gebruikt:
private string GetSqlType(ExpressionType exprType)
   {
      switch (exprType)
      {
          case ExpressionType.GreaterThanOrEqual: return ">=";
          case ExpressionType.AndAlso: return "AND";
          case ExpressionType.LessThan: return "<";
                //etc
      }
}
```

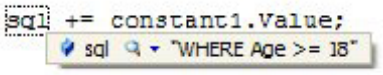
En om toe te voegen aan de sql query:

```
sql += " " + GetSqlType(left.NodeType) + " ";
```

Het enige wat we voor de linkerkant hoeven doen, is de verwijzing naar '18'. Omdat dit gewoon een constante is, is dit erg eenvoudig:

```
sql += constant1.Value;
```

Als we nu de code uitvoeren, zien we dat de linkerkant van de boom volledig geconverteerd is naar een SQL string.



Nu moeten we de rechterkant van de boom ook toevoegen. Eerst voegen we het type van de vergelijking tussen de linker en de rechtertak van de 'AndAlso' boom toe:

```
sql += " " + GetSqlType(binary.NodeType) + " ";
```

Voor de rechterkant van de boom, kopiëren we ons eerdere stuk code, en passen alleen 'left' aan naar 'right'.

```
BinaryExpression right = (BinaryExpression)binary.Right;
MemberExpression property2 = (MemberExpression)right.Left;
ConstantExpression constant2 = (ConstantExpression)right.Right;

sql += property2.Member.Name;
sql += " " + GetSqlType(right.NodeType) + " ";
sql += "'" + constant2.Value + "'";
</code>
Bekijken we nu de waarde van 'sql', dan zien we een SQL query die
klaar is om te gebruiken.
<invoegen espression trees6.jpg>
Omdat dit allemaal pas gemaakt wordt, als de expressie wordt ge-
parset, hoeven we alleen maar de expressie aan te passen, om toch
een goede SQL query te krijgen.
<code>
Expression<Func<Person, bool>> personDelegate = person => person.
Age < 20 && person.City == "Zaandam";
```

Geeft bijvoorbeeld netjes het volgende terug:

```
WHERE Age < 20 AND City = 'Zaandam'
```

En voila: we hebben een dynamische expressie omgezet naar een bruikbare SQL query. Voordelen hiervan zijn:

- Geen syntaxfouten meer mogelijk. Er wordt namelijk compile time checking gedaan op je queries;
- Strongly typed queries; dus intellisense etctera;
- Elke domain specific language kan worden getarget met slechts een syntax, en een andere parser.

Omdraaien

Een expressie tree zelf heeft ook een aantal voordelen. Deze kan namelijk opgebouwd worden in code (denk aan, het uitlezen van een SQL string en dit omzetten naar een boom). Daarnaast kan een expressie tree worden gecompileerd naar een delegate, die vervolgens talloze keren kan worden hergebruikt in een applicatie, waarbij er maar een keer de omzetslag van abstracte weergave naar uitvoerbare code hoeft worden gemaakt. Voor een voorbeeld bouwen we de boom uit het vorige voorbeeld eens op vanaf de andere kant:
Omdat we gebruik gaan maken van verwijzingen naar properties, moeten we eerst zorgen dat we een parameter kunnen gebruiken in de boom, van het type 'person'.

```
ParameterExpression parameter = Expression.
Parameter(typeof(Person), "person");
```

Als we in het vervolg willen verwijzen naar een property op een 'person' object, gebruiken we 'parameter' als referentie. We beginnen aan de linkerkant van de boom, en wel vanaf de onderkant. Hier hebben we een 'MemberExpression' (naar per-son.Age), en een 'ConstantExpression':

```
MemberExpression personAge = Expression.Property(parameter,
"Age");
ConstantExpression eightteen = Expression.Constant(18);
```

Samen worden deze gecombineerd tot een 'BinaryExpression' met als vergelijking 'GreatherThan':

```
BinaryExpression left = Expression.GreaterThanOrEqual(personAge,
eightteen);
```

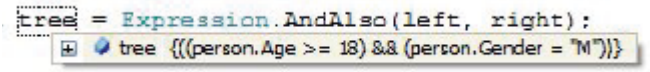
Als we nu over 'left' heengaan tijdens het uitvoeren van het programma, dan zien we dat de linkerkant van de boom perfect is opgebouwd:

We doen nu hetzelfde voor de rechterkant van de boom:

```
MemberExpression personGender = Expression.Property(parameter,
"Gender");
ConstantExpression m = Expression.Constant("M");

BinaryExpression right = Expression.Equal(personGender, m);
```

Nu hoeven we alleen 'left' en 'right' nog te combineren, om de boom compleet te hebben:

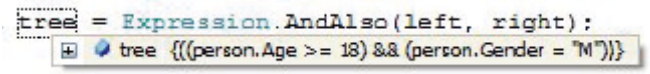


Compileren

Om de tree om te zetten naar programmacode, kan de compiler de boom omzetten naar een delegate, die we vervolgens kunnen gebruiken om objecten te testen tegen de regels die we net hebben ingevoerd. Hiervoor moeten we de expressie wrappen in een lambda expressie. Hierin geven we ook de referentie naar de ParameterExpression mee, zodat we deze kunnen meegeven aan de delegate straks:

```
LambdaExpression lambda = Expression.Lambda(tree, parameter);
```

Wat resulteert in een nette lambda expressie:



Exact hetzelfde als we eerder handmatig hebben gedaan. Wanneer we de Compile() functie aanroepen kunnen we deze code nu gebruiken als normale delegate, aan wie we eender welk person object geven.

```
Func<Person, bool> persDelegate = (Func<Person, bool>)lambda.Com-
pile();

bool validates = persDelegate.Invoke(new Person() { Age = 19, Gen-
der = "M" });
```

En 'validates' bevat nu 'true'. Om een vergelijking te maken, hoeveel sneller iets als reflection op deze manier kan; vergelijk ik het testen van 1.000.000 objecten via reflection, en via een expression tree:

Via de expression tree

```
//opbouwen van de expression tree

Func<Person, bool> persDelegate = (Func<Person, bool>)lambda.Com-
pile();

Person p = new Person() { Age = 21, Gender = "F" };
for (int i = 0; i < 1000000; i++)
   persDelegate.Invoke(p);
```

Via reflection

```
PropertyInfo ageProperty = typeof(Person).GetProperty("Age");
PropertyInfo genderProperty = typeof(Person).GetProperty("Gender");

Person person = new Person() { Age = 21, Gender = "F" };

for (int i = 0; i < 1000000; i++)
{
    if (((int)ageProperty.GetValue(person, null)) > 18
     && ((string)genderProperty.GetValue(person, null)) == "M")
      { }
}
```

Native

```
Func<Person, bool> personDelegate = person2 => person2.Age > 18 &&
person2.Gender == "M";

for (int i = 0; i < 1000000; i++)
personDelegate.Invoke(p);
```

Resultaten

Expression tree	83 ms
Reflection	2903 ms
Native	81 ms

Zoals duidelijk te zien is, kost het gebruik van de expression tree nauwelijks overhead, en is er een ongelooflijk voordeel ten opzichte van reflection. Dynamische intensieve taken zijn dan ook uitermate geschikt om te gebruiken door middel van een expression tree. De snelheid en cachebaarheid van de delegates maakt het bijvoorbeeld ongelooflijk goed schaalbaar om business objecten om te zetten via reflection, wat ervoor zorgt dat ontwikkelaars niet weken bezig zijn met translator classes, maar de code het werk kunnen laten doen.

Conclusie

In dit artikel hebben we gezien hoe expression trees werken, hoe ze te gebruiken zijn om de voordelen van het schrijven van C# code kunnen gebruiken om te praten via DSL's als SQL; en hoe we expression trees kunnen gebruiken om veelvoorkomende dynamische intensieve taken te versnellen. Voor een bruikbaar stuk code hoe reflection te omzeilen met expression trees, heeft Jon Skeet een voorbeeld geschreven, de code is te vinden op: http://stackoverflow.com/questions/531505/how-to-copy-value-from-class-x-to-class-y-with-the-same-property-name-in-c

Jan Jongboom, is medewerker bij de plantsoenendienst. In zijn vrije tijd maakt hij funda.nl.