There Be Dragons

By Bill Wagner
June 2011

Examples of problems

It's often been written that declarative programs specify what you want, where imperative programs specify how you want it done. Mixing these concepts incorrectly can cause serious problems. A couple small examples will illustrate. First up, examine this query:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

int tmp;

var nums = from item in seq

    let success = int.TryParse(item, out tmp)

    select success ? tmp : 0;
```

You may be surprised to learn that he above snippet does not compile. The compiler reports that you may be using an initialized variable, tmp. That, of course, is easy to fix. However, a few small changes produce more insidious errors:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

int tmp = 0; // eliminate the uninitialized variable bug

var nums = from item in seq

    where int.TryParse(item, out tmp)

    orderby tmp

    select tmp;
```

You may expect this to produce the sequence { 1, 2, 3, 5 }, but it doesn't. It produces a completely different result. In the rest of this article, you'll learn why these constructs produce the results they do, and what you can do about it.

Why the compilation error occurs

Let's start with that first query. How can the compiler possibly think that tmp isn't definitely assigned? The let expression calls TryParse, which guarantees (by using the out modifier) that tmp will be

definitely assigned before TryParse exits. In fact, re-writing the original query to use an imperative style works:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

int tmp;

var answer = new List<int>();

foreach (var item in seq)

   answer.Add(int.TryParse(item, out tmp) ? tmp : 0);
```

The sequence answer will contain the expected sequence: { 1, 2, 3, 5}. So why doesn't the query expression compile? Let's rewrite the query as the following equivalent method calls:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

int tmp;

var nums = seq.Select(item => new

{

   item,

   success = int.TryParse(item, out tmp)

})

.Select(transparent => transparent.success ? tmp : 0);
```

The compiler maps query expressions onto the query expression pattern's method calls. See section 7.16.3 of the C# Language Specification, or Item 36 in More Effective C# for more details on the query expression pattern. Those two code constructs are effectively the same. The key to understanding the problem is the first Select call. Notice that the parameter to Select is a lambda expression that will produce a new object when requested. There's no guarantee that Select() will ever actually execute that lambda expression and produce an item. There's no guarantee that TryParse() will ever be called. If TryParse isn't called, tmp isn't initialized. From a human perspective, it may seem reasonable to expect that Select will call the lambda expression it's given. However, many reasonable implementations may not. It may store a copy of the lambda for execution later. Select may just ignore its parameter entirely. The compiler does the correct analysis here.

I do not want you to get the impression that this issue is because of query expressions. This behavior will occur anytime you create a lambda expression or anonymous method that assigns an outer variable. Section 5.3.3.29 of the C# Language Specification defines this behavior. Defining an anonymous method or lambda expression does not execute the lambda, therefore it does not change the definite assignment state of any variable referenced inside the body of the method or expression.

Why the runtime error occurs

If the second example doesn't produce the correct sequence, what is its output? It produces the sequence { 1, 1, 1, 1 }. Does that information give you a clue to the reason for the problem? Why does the query return the last item in the sequence four times? Once again, the answer involves the side effects that are part of TryParse. One of the implementation details for Enumerable.OrderBy() is that it moves the Enumerator used in the Where call to its end. The OrderBy is asked for the first item, it must examine the entire sequence in order to determine the smallest item. Before it returns any item, it must enumerate its entire input sequence. By the time you return any of the items, tmp contains the value calculated for the last item in the sequence. All the side effects for the entire input sequence have happened before OrderBy returns its first value.Again, if we re-write the query as its equivalent method calls it becomes a little clearer:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

int tmp = 0; // eliminate the uninitialized variable bug

var nums = seq.Where(item => int.TryParse(item, out tmp)).

   OrderBy(item => tmp).Select(item => tmp);
```

Now the error is more clear. The OrderBy method will order by tmp. However, by the time OrderBy determines the order of the sequence, tmp has the value '1'. Every value computes to the same order value. In the same manner, the last Select will return tmp for each and every item in its input sequence. It's just the same value everytime.

As with the early problem found by the compiler, this is not because you chose to use query expressions. Rather, it is because you were relying on side effects, and those side effects occurred at a different time than you expected. You specified how; the compiler thought you were specifying what.

Lessons To Learn

To fix these problems you need to modify how TryParse returns its information. Look at the signature of int.TryParse:

```
public static bool TryParse(
        string s,
        out int result
)
```

It doesn't return the integer value it found. Rather, it modifies a storage location provided to it. It isn't a pure function: it modifies program state. Pure functions can neither write to external state, nor read from external mutable state. They must always provide the same output for the same input, and they must not produce any observable side effect. When TryParse executes, it will modify the storage location referred to by tmp. That's an observable side effect. This is pure evil in a query expression. To understand why, you need to understand how the compiler transforms a query expression into

executable IL, and we'll need to work with the terms 'closure', 'bound variable' and 'free variable'. I'm going to take a few liberties and simplify the definitions a little.

A bound variable is a formal argument to a function. When you write:

Func<int, int> foo = x => x * 2;

x is a bound variable for that lambda expression. When you call foo(5), you have bound the value of 5 to x.

A free variable is a local variable or formal variable declared outside of the lambda. Foo, as defined above, does not have any free variables. This definition of bar does:

int i = 2;

Func<int, int> bar = x => x * i;

The variable i is a free variable. Its value will be read when bar() is executed. For example, consider this sequence of code:

int i = 2;

Func<int, int> bar = x => x * i;

var q = bar(3); // q is 6

i = 4;

q = bar(3); // q is 12

i = 8;

q = bar(3); // q is 24

i = 2;

q = bar(3); // q is 6

The compiler creates a closure to implement this behavior. A closure is a function along with an environment for all free variables of that function. The closure contains the variable i, not the value of i when the closure is created.

Armed with those three working definitions we can see what the compiler creates, and why this code doesn't do what we'd hoped.

When you define a lambda expression like foo above, the compiler creates a static method, and a delegate that executes that method. :

For the lambda expression bar, the compiler must generate much more code. A static method won't work. Where can it access the current value of i? The compiler must create a closure that contains the

function, along with storage for the current value of i. That sounds like a class containing one field (i), and one method (the code for the lambda expression). That's what the compiler creates. It looks something like this:

```
private sealed class Generated
{
   public int i;
   public int LambdaForBar(int x)
   {
      return (x * this.i);
   }
}
```

Now you can see why the query expression doesn't do what you think. You're creating side-effects in a closed over variable. The value of that closed over variable (tmp) changes throughout the execution of the query. However, the query relies on its value at some previous point in time.

Now that we understand what's going on, let's fix these problems. The first step is to write a small extension for TryParse that returns its value, rather than introducing side-effects:

```
public static int? TryParse(this string s)
{
   int tmp;
   return int.TryParse(s, out tmp) ?
      tmp :
      default(int?); // null, typed as int?
}
```

This new version of TryParse returns its result, rather than modifying some other external storage. Now, we can rewrite our original examples to use this new method.

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };
var nums = from item in seq
         select item.TryParse() ?? 0;
```

and:

```
var seq = new List<string> { "2", "5", "not a number", "3", "four", "1", };

var nums = from item in seq
        let num = item.TryParse()
        where num.HasValue
        orderby num.Value
        select num.Value;
```

In both these cases, the code does not introduce a free variable. It especially does not introduce side effects on a closed over variable. They no longer introduce any kind of side effects at all.

I used int.TryParse() as my examples in this article because it's fairly simple to understand, and does introduce side effects. It has the signature it has because of when it was written. It was written before generics, before tuples, before extension methods, and before query expressions. These problems couldn't exist. You may very well be working with similar APIs from a different time. As you work with these older APIs, you may find yourself with similar unexpected behavior. In those cases, you'll want to take similar action: make a new API that removes the side effect from the query.

In the modern world, we'll use multiple programming concepts at different times because they are the best tool for the job. However, mixing these constructs can cause wrong behavior. Introducing side effects into queries causes problems when those query expressions rely on side-effects at a point in time. As a professional developer, it is important to understand the reasons behind common recommendations. That gives you the tools to work across multiple idioms safely.