Making the code read like the spec

by ERIC LIPPERT, XOPOSHIY, BRIAN, GABE, GRICO, ROBERT DAVIS, SZYMON KULEC, NICK, JON SKEET, THOMAS LEVESQUE, BLAKE, ROB, JEFF YATES, STUART BALLARD, PAVEL MINAEV | FEB. 8, 2010

Профиль завершен Ответ с кодом I Знаток форумов I

As I mentioned a while back, there are some bugs in the compiler code which analyzes whether a set of classes violates the "no cycles" rules for base classes. (That is, a class is not allowed to inherit from itself, directly or indirectly, and not allowed to inherit from one of its own nested classes, directly or indirectly.) The bugs are almost all of the form where we accidentally detect a cycle in suspicious-looking generic code but in fact there is no cycle; the bugs are the result of an attempt to modify the cycle detector from C# 1 rather than simply rewriting it from scratch to handle generics. Unfortunately we were unable to get the fixes into C# 4; these are obscure corner-case scenarios and the risk of doing the fix was considered too high.

There are a number of tricky issues here, mostly around the fact that obviously we cannot know whether a set of base types are circular until we know what the base types are, but resolving the program text to determine what type the base type string "C.D<E.F>" requires us to know the base types of C, because D might be a nested type of C's base class, not of C, so we have a bit of a chicken-and-egg problem. The code which turns strings into types has to be robust in the face of circular base types because the base type cycle detector depends on its output!

So like I said, I've come up with a new algorithm that implements the spec more exactly, and I wanted to test it out. Rather than modifying the existing compiler to use it, I mocked it up in C# quickly first, just to give me something to play with. One of the problems that we have with the existing compiler is that it is not at all clear which parts of the code are responsible for implementing any given line in the spec. In my "maquette" of the compiler I wanted to make sure that I really was exactly implementing the spec; that might show up logical problems with either the implementation or the spec. I therefore wanted the code to read much like the spec.

This little hunk of code that I wrote made me inordinately happy. Here's the spec:

A class *directly depends on* its direct base class (if any) and *directly depends on* the class within which it is immediately nested (if any). The complete set of classes upon which a class *depends* is the *reflexive and transitive closure* of the directly-depends-upon relationship.

First off, what is this thing called the “reflexive and transitive closure”?

Consider a “relation” – a function that takes two things and returns a Boolean that tells you whether the relation holds. A relation, call it ~>, is *reflexive* if X~>X is true for every X. It is *symmetric* if A~>B necessarily implies that B~>A. And it is transitive if A~>B and B~>C necessarily implies that A~>C. (*)

For example, the relation “less than or equal to” on integers is reflexive: X≤X is true for all X. It is not symmetric: 1≤2 is true, but 2≤1 is false. And it is transitive: if A≤B and B≤C then it is necessarily true that A≤C.

The relation “is equal to” is reflexive, symmetric and transitive; a relation with all three properties is said to be an “equivalence relation” because it allows you to partition a set into mutually-exclusive “equivalence classes”.

The relation “is the parent of” on people is not reflexive: no one is their own parent. It is not symmetric: if A is the parent of B, then B is not the parent of A. And it is not transitive: if A is the parent of B and B is the parent of C, then A is not the parent of C. (Rather, A is the grandparent of C.)

It is possible to take a nontransitive relation like “is the parent of” and from it produce a transitive relation. Basically, we simply make up a new relation that is exactly the same as the parent relation, except that we *enforce* that it be transitive. This is the “is the ancestor of” relation: if A is the ancestor of B, and B is the ancestor of C, then A is necessarily the ancestor of C. The “ancestor” relation is said to be the transitive closure of the “parent” relation.

Similarly we can define the reflexive closure, and so on.

When we’re talking about closures, we’re often interested not so much in the relation itself as the *set of things which satisfy the relation with a given item*. That’s what we mean in the spec when we say “The complete set of classes upon which a class *depends* is the *reflexive and transitive closure* of the directly-depends-upon relationship.” Given a class, we want to compute the set that contains the class itself (because the closure is reflexive), its base class, its outer class, the base class of the base class, the outer class of the base class, the base class of the outer class, the outer class of the outer class... and so on.

So the first thing I did was I wrote up a helper method that takes an item and a function which identifies all the items that have the non-transitive relation with that item, and computes from that the set of all items that satisfy the transitive closure relation with the item:

```
static HashSet<T> TransitiveClosure<T>(
   this Func<T, IEnumerable<T>> relation,
   T item)
{
```

```
    var closure = new HashSet<T>();
    var stack = new Stack<T>();
    stack.Push(item);
    while(stack.Count > 0)
    {
        T current = stack.Pop();
        foreach(T newItem in relation(current))
        {
            if (!closure.Contains(newItem))
            {
                closure.Add(newItem);
                stack.Push(newItem);
            }
        }
    }
    return closure;
}
static HashSet<T> TransitiveAndReflexiveClosure<T>(
    this Func<T, IEnumerable<T>> relation,
    T item)
{
  var closure = TransitiveClosure(relation, item);
  closure.Add(item);
  return closure;
}
```

Notice that essentially what we're doing here is a depth-first traversal of the graph defined by the transitive closure relation, avoiding descent into regions we've visited before.

Now I have a mechanism which I can use to write code that implements the policies described in the specification.

```
static IEnumerable<TypeSymbol> DirectDependencies(TypeSymbol symbol)
{
    // SPEC: A class directly depends on its direct base class (if any) ...
    if (symbol.BaseTypeSymbol != null)
        yield return symbol.BaseTypeSymbol;
    // SPEC: ...and directly depends on the class within which it
    // SPEC: is immediately nested (if any).
    if (symbol.OuterTypeSymbol!= null)
        yield return symbol.OuterTypeSymbol;
}
```

Great, we now have a method that exactly implements one sentence of the spec – given a type symbol, we can determine what its direct dependencies are. Now we need another method to implement the next sentence of the spec:

```
static HashSet<TypeSymbol> Dependencies(TypeSymbol classSymbol)
{
  // SPEC:  The complete set of classes upon which a class
  // SPEC:  depends is the reflexive and transitive closure of
  // SPEC:  the directly-depends-upon relationship.
  return TransitiveAndReflexiveClosure(DirectDependencies, classSymbol);
}
```

That's what I like to see: code that reads almost exactly like the spec.

Note also that I've made the design choice here to make these methods static methods of a policy class, rather than methods on the TypeSymbol class. I want to keep my policies logically and textually separated from my mechanisms. For example, I could be using the same classes that represent classes, structs, namespaces, and so on, to implement VB instead of C#. I want the policies of the C# language to be in a class whose sole responsibility is implementing these policies.

Another nice aspect of this approach is that I can now re-use my transitive closure computing mechanism when I come across this bit of the spec that talks about base interfaces of an interface:

The set of base interfaces is the transitive closure of the explicit base interfaces.

Unsurprisingly, the code in my prototype that computes this looks like:

```
static HashSet<TypeSymbol> BaseInterfaces(TypeSymbol interfaceSymbol)
{
  // SPEC: The set of base interfaces is the transitive closure
  // SPEC: of the explicit base interfaces.
  return TransitiveClosure(ExplicitBaseInterfaces, interfaceSymbol);
}
```

In fact, there are transitive closures in a number of places in the C# spec, and now I have a mechanism that I can use to implement all of them, should I need to for my prototype.

One final note: notice that I am returning a mutable collection here. Were I designing these methods for public consumption, I would probably choose to return an immutable set, rather than a mutable one, so that I could cache the result safely; these methods could be memoized since the set of dependencies does not change over time. But for the purposes of my quick little prototype, I'm just being lazy and returning a mutable set and not caching the result.

Hopefully we'll get this new algorithm into the hypothetical next compiler release.

****

(*) Incidentally, we are holding on to the wavy arrow ~> as a potential operator for future hypothetical versions of C#. We recently considered it to have the meaning a~>b() means ((dynamic)a).b(). The operator is known as “the naj operator” after Cyrus Najmabadi, who advocated the use of this operator to the C# design team. Anyone who has a great idea for what the naj should mean, feel free to leave a comment.

C#, Code Quality, transitive closure

Original URL: http://blogs.msdn.com/b/ericlippert/archive/2010/02/08/making-the-code-read-like-the-spec.aspx