Treecc: An Aspect-Oriented Approach to Writing Compilers

Rhys Weatherley
rweather@southern-storm.com.au

Jan 10, 2002

1 Introduction

The C# compiler in Portable.NET [1] is built on top of the **"Tree Compiler Compiler"** (treecc) utility program. Treecc [2] is distributed as Free Software under the terms of the GNU General Public License.

This article provides some background of why treecc came about. It discusses two common compiler implementation techniques, and the reasons why they often fail to manage the complexity of large programming languages like C#.

We then discuss a new programming technique called **"Aspect-Oriented Programming"**. Treecc is an example of applying this technique to managing the complexity of compiler construction.

2 Patterns and Compiler Design

Compiler writing is generally seen as a black art, but in reality it isn't all that hard. The basic compilation steps are:

1. *Convert the program into an abstract syntax tree.*
2. *Perform type-checking and semantic analysis on the tree.*
3. *Rearrange the tree to perform optimisations.*
4. *Convert the tree into the target code.*

The difficulty in writing compilers is not the steps involved, but rather the sheer number of tiny little details to keep straight. Modern languages contain large numbers of operators, which are all very similar, but slightly different:

```
Add, substract, multiply, divide,
remainder, shift left, shift right, shift
right unsigned, bitwise and, bitwise or,
exclusive-or, negate, bitwise not, logical
not, logical and, logical or, ...
```

And that's just the operators. Introduce statements, arrays, type coercion, method invocation, and class definition, and it becomes very easy to forget something amongst the forest of code.

In an attempt to control this complexity, two common pattern-based approaches have arisen over the years: Inheritance and Visitor.

The inheritance pattern can be characterised as follows:

1. Declare a node type for every syntactic element in the language. All types ultimately inherit from ``Node''.
2. Declare virtual methods in ``Node'' for operations on node types: semantic analysis, opti-

[1] Portable.NET Web Site, *http://www.southern-storm.com.au/portable_net.html.*
[2] Treecc Web Site, *http://www.southern-storm.com.au/treecc.html.*

mization, code generation, etc.

3. Override the virtuals in sub-classes to provide the compiler implementation.

The visitor pattern can be characterised as follows:

1. Declare a node type for every syntactic element in the language. All types ultimately inherit from ``Node''.

2. Declare a ``Visitor'' class with abstract virtual methods such as ``VisitAdd'',``VisitSub'', ``VisitIf'', ``VisitFunction'', etc. for all of the node types.

3. Define a ``walking procedure'' over ``Node'' objects for walking around a syntax tree, making callbacks on a supplied visitor object.

4. Create multiple classes that inherit from ``Visitor'', one for each operation. Implement the ``Visit*'' functions for that type of operation.

In the following sections, we will explore why these two patterns often fail to manage compiler complexity, even when the programmer applies them rigorously.

3 The implementation language is your worst enemy

We will start with the inheritance pattern. Consider that we've written the following C# code during the *"declare all the node types"* phase of the project:

```
public class UnaryExpression : Expression
    {
        protected Expression expr;

        public UnaryExpression(Expression _expr)
            { expr = _expr; }
    }

    public class NegateExpression : UnaryExpression
    {
        public NegateExpression(Expression _expr)
            : base(_expr) {}
    }

    public class UnaryPlusExpression : UnaryExpression
    {
        public UnaryPlusExpression(Expression _expr)
            : base(_expr) {}
    }

    public class BitwiseNotExpression : UnaryExpression
    {
        public BitwiseNotExpression(Expression _expr)
            : base(_expr) {}
    }
    }
```

We continue this process for several hundred other node types, gradually building up the entire syntax tree. This will probably take several weeks to complete for a substantial language like C#, assuming that we are writing the parser alongside the node types.

We now want to go in and implement type-checking, so we modify the "UnaryExpression" class as follows:

DotGNU is a Free Software project to create a platform for webservices that can be written in a variety of different programming languages including Java and C#.

As Microsoft is trying to catch all e-commerce in their .NET and lock everyone in. so we at FSM believe DotGNU is a very important project to protect our community, thus we have set this column for it. With the support of DotGNU Project developers, we will bring you more articles about DotGNU core technologies in the future issues.

```
public class UnaryExpression : Expression
    {
        protected Expression expr;

        public UnaryExpression(Expression _expr)
            { expr = _expr; }

        public override LanguageType TypeCheck()
        {
            LanguageType type = expr.TypeCheck();
            if(type.IsInteger() || type.IsFloat())
            {
                return type;
            }
            throw new TypeCheckException();
        }
    }
```

We've put the common unary expression type-checking code in a common base class. This makes it easier to maintain because there is only one copy. We continue the process over the next few weeks and months for all the other operators, statements, and declarations in the language. So far, so good.

Unfortunately, we've made a mistake. The `"BitwiseNot"` operator is only legal on integer values; not floating-point.

But will we find this bug? It was several weeks ago when we first wrote the `"BitwiseNotExpression"` class, and we have since forgotten all about it. It may even have been written by another programmer on the team, who has also forgotten all about it. When we build our compiler, we don't get any errors because the implemention language is perfectly happy with the above code.

Surely testing will find it? We are building a test suite alongside the code, right? Unfortunately, that doesn't help either. The test suite for a major language will be at least as complex as the compiler itself, and so there is always the temptation to abstract common tests into common test classes. We've just shifted the bug into the test suite and given ourselves a false sense of security.

So we keep coding for several more months, adding lots more code. And then a really nasty bug pops up. The `"BitwiseNot"` operator is acting strangely, and we have no idea why. The system is now so complex, with so many common base classes implementing fallback defaults, that tracking this down becomes very hard.

The inheritance pattern has a fatal flaw. Adding a new operation entails a very large maintainence burden, because hundreds of classes must be modified. This is very error-prone, so we try to abstract details into common base classes. But this introduces other errors.

The problem basically boils down to semantic analysis: the implementation language does not have enough knowledge about the application domain to spot the problem and warn us about it. So it happily compiles the code and leaves us to hang ourselves on the system's complexity later. Programming languages are supposed to help us manage complexity, not make the problem worse!

I wrote a number of compilers using the inheritance approach, and every single time the complexity killed me. I needed fallback defaults for code maintainence reasons, but using fallbacks introduced massive numbers of bugs. I was stuck.

4 Design Patterns Aren't Always What They Are Cracked Up to Be

After much hair-pulling, I searched Design Patterns by Gamma, et al [3]. "Is there a better way of doing this?". Visitor patterns are the answer: the book even gives a compiler example.

Visitors solve the "I forgot to implement the

[3] Gamma, et al., **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995.

`BitwiseNot`" problem. Because every node type has its own ``Visit*'' method, we will get an error when we try to build the compiler without implementing the operation for a node type. Of course, this assumes that we haven't been dumb and implemented fallback defaults in the ``Visitor'' base class.

Instead of using virtual methods, we can implement visitors using switch statements over node types. e.g.

```
switch(node.type)
{
    case Negate: ...
    case UnaryPlus: ...
    case BinaryNot: ...
    ...
}
```

However, switch statements have a similar flaw to inheritance: the implementation language will not warn us if we forget to put in a case for every node type. It will happily fall out through the "default" case with no warning. Tracking down these bugs can be just as hard as tracking down inheritance fallback bugs. Using virtual methods is "safer", if not quite as efficient.

Unfortunately, there is a catch with visitors, as explained in *Design Patterns*:

Use the Visitor pattern when ... the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it's probably better to define the operations in those classes.

During the early stages of writing a compiler, the node types change very frequently. This activates the Achilles heel of the Visitor pattern, and creates a maintainence nightmare. The book suggests that we should use the inheritance approach to solve this problem.

5 So Which One Do We Use? Inheritance or Visitor?

The inheritance pattern becomes a problem when new operations are needed. The solution is visitors. Visitors become a problem when new node types are needed. The solution is inheritance.

What we have is a situation that the design patterns gurus didn't consider: if the set of nodes and operations are both changing rapidly, then we will have problems no matter what we do.

We need a solution that combines the strengths of both patterns without the drawbacks of either. We want the implementation language to catch us when we forget something, but we also want it to handle large numbers of nodes and operations smoothly. We want to split different operations into different modules, but also keep them closely associated with the node type.

None of the standard patterns provide this combination of functionality, because none of the existing implementation languages support both styles of program design at the same time.

6 Aspect-Oriented Programming

A new field in language design has emerged in recent years called **"Aspect-Oriented Programming"** (AOP). A good review of the field can be found in the October 2001 issue of the **"Communications of the ACM"** [4], and on the AspectJ Web site [5].

The following excerpt from the introduction to the AOP section in the CACM issue describes the essential aspects of AOP, and the difference between OOP and AOP:

[4] Aspect-Oriented Programming, Communications of the ACM, October 2001.

[5] AspectJ Web Site, *http://www.aspectj.org/*.

AOP is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties or areas of interest) of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. ... While the tendancy in OOP's is to find commonality among classes and push it up the inheritance tree, AOP attempts to realize scattered concerns as first-class elements, and eject them horizontally from the object structure.

Aspect-orientation gives us some hope of solving our compiler complexity problems. When we moved from the inheritance pattern to the visitor pattern, we were attempting to eject the operations horizontally. But it didn't quite work as well as we had hoped: the intrinsic complexity of the set of nodes kept interfering. AOP allows us to take the idea further, without re-introducing the problems that visitors have.

We can view each operation on node types (semantic analysis, optimization, code generation, etc) as an "aspect" of the compiler's construction. The AOP language weaves these aspects with the node types to create the final compiler.

However, we don't really want to invent a completely new programming language for compiler construction. We would have to implement this new language using all of the flawed techniques that makes writing compilers hard. It's a classic chicken and egg problem: we don't want to replace a buggy compiler with a buggy compiler implementation language.

We can strike a compromise, similar to that used by lex and yacc. Those tools use a custom syntax for the difficult parts, and a pre-existing underlying language (usually C) to implement everything else. The code is expanded by the tool and then compiled with the underlying language's compiler.

Treecc uses a domain-specific aspect-oriented programming language for declaring and managing nodes and operations, and uses an underlying language to implement the body of the operations. C, C++, C#, or Java can be used as the underlying language, depending upon your personal preference.

Treecc is about 13,000 lines of code in size, which is relatively easy to debug by hand.

Aside: treecc does not support all of the AOP features that are described in the literature. Treecc weaves together classes from multiple method definitions. Other AOP languages can also weave together methods from fragments in multiple aspects. We concentrated on those AOP features that were useful for compiler construction. Other features could be incorporated at a later date.

7 Using Treecc to Beat Inheritance Bugs

Now that we've identified the problems of inheritance and visitor patterns for compiler implementation, we will show how treecc helps the programmer avoid these traps.

The following is the treecc definition of our example node types:

```
%option lang = "C#"

%node Expression %abstract %typedef

%node UnaryExpression Expression %abstract =
{
    Expression expr;
}

%node NegateExpression UnaryExpression
%node UnaryPlusExpression UnaryExpression
%node BitwiseNotExpression UnaryExpression
```

Treecc converts this into a number of C# classes, one for each node type. It also inserts helper methods for testing the type of a node and for tracking source line numbers. If the output language is C or C++, treecc will also insert code for allocating large numbers of

nodes efficiently.

The type-checking operation (or "aspect") is declared as follows:

```
%operation %virtual LanguageType
                  TypeCheck(Expression e)

    TypeCheck(NegateExpression),
    TypeCheck(UnaryPlusExpression)
    {
        LanguageType type = e.expr.TypeCheck();
        if(type.IsInteger() || type.IsFloat())
        {
            return type;
        }
        throw new TypeCheckException();
    }

    TypeCheck(BitwiseNotExpression)
    {
        LanguageType type = e.expr.TypeCheck();
        if(type.IsInteger())
        {
            return type;
        }
        throw new TypeCheckException();
    }
```

We have not declared the operation to cover ``Expression'' or ``UnaryExpression''. Instead, we list all of the applicable subtypes explicitly. When treecc is run on the above file, it will check that every non-abstract node type is handled by an operation case. If it finds a missing type, it will report an error. Let's demonstrate that by adding a new unary expression type:

```
%node LogicalNotExpression UnaryExpression

    eg.tc:17: node type 'LogicalNotExpression' is not
    handled in operation 'TypeCheck'
```

This is at the heart of treecc's power: it performs a large amount of semantic analysis over the node types to determine if the programmer has forgotten something. The programmer is notified of this early in development process, when it is easier to fix the problem.

As we discussed earlier, we want to implement common code in common base classes to improve code sharing. However, this introduces hard to find bugs. Treecc avoids the need to do this by allowing the programmer to attach multiple cases to the same code block, as in the case of ``NegateExpression'' and ``UnaryPlusExpression'' above.

An important feature of aspect-oriented languages is aspect modularity: it should be possible to implement separate aspects in different parts of the code. Treecc supports this by separating node and operation definitions. Operations do not need to be implemented in the same file as the nodes to which they apply, and multiple operations on the same node types can be scattered through-out the code.

Portable.NET takes this even further by separating individual operations across multiple files for expressions, statements, declarations, etc. This introduces a clearer structure to the code that makes it easier to navigate the source during compiler construction. The semantic analysis routines in treecc ensure that nothing is missed when all of the separate modules are recombined.

8 Painless Visitors

The previous example used a ``%virtual'' operation, which is defined over all node types. Treecc inserts the virtual method body wherever it is required.

Sometimes we don't want to define an operation as a virtual method. We would prefer to use the visitor approach. The following is what a visitor version of "TypeCheck" would look like:

```
%operation LanguageType
       TypeChecker::TypeCheck(Expression e)
       = {null}

TypeCheck(NegateExpression),
TypeCheck(UnaryPlusExpression)
{
    LanguageType type = TypeCheck(e.expr);
    if(type.IsInteger() || type.IsFloat())
    {
        return type;
    }
    throw new TypeCheckException();
}

TypeCheck(BitwiseNotExpression)
{
    LanguageType type = TypeCheck(e.expr);
    if(type.IsInteger())
    {
        return type;
    }
    throw new TypeCheckException();
}
```

This is almost identical to the previous version, except for the definition of the operation, and the calling conventions for "`TypeCheck`". Behind the scenes, treecc creates a class called "`TypeChecker`" that contains a static method called "`TypeCheck`". This is the visitor.

Interestingly, if we had used C as the underlying language, then no changes are necessary to the bodies of the operation cases. Only the "`%virtual`" keyword changes. The C macro pre-processor is used to smooth out the differences.

This illustrates another useful property of treecc: it is very simple to flip operations between inhertance-based virtuals and visitor-based non-virtuals. This allows the programmer to start developing the compiler one way, change their mind, and quickly flip to the other way.

Normally, changing inheritance-based code into visitor-based code would entail a complete system rewrite. Changing patterns with treecc is trivial.

This is a common property of aspect-oriented programming languages: because the language takes care of the inserting the aspects into the main classes, it is easier to change the style of insertion without a major system overhaul.

Non-virtual operations can be applied to multiple arguments, which can be very useful when implementing coercions:

```
%enum SimpleType =
    {
        Integer,
        Long,
        Float,
        Error
    }

    %operation %inline SimpleType Binary::Coerce
            ([SimpleType type1], [SimpleType type2])
            = {Error}

    Coerce(Integer, Integer)
    {
        return Integer;
    }

    Coerce(Integer, Long),
    Coerce(Long, Integer),
    Coerce(Long, Long)
    {
        return Long;
    }

    Coerce(Integer, Float),
    Coerce(Float, Integer),
    Coerce(Long, Float),
    Coerce(Float, Float)
    {
        return Float;
    }

    Coerce(Integer, Error),
    Coerce(Error, Integer),
    Coerce(Long, Error),
    Coerce(Error, Long),
    Coerce(Float, Error),
    Coerce(Error, Float),
    Coerce(Error, Error)
    {
        return Error;
    }
```

Treecc turns this into a highly efficient nested switch statement, which would be extremely difficult to de-

bug by hand. We actually left out one of the cases above, so we get an error:

```
eg.tc:48: case 'Float, Long' is missing from
     operation 'Coerce'
```

Casts and coercions on primitive types can now be implemented as simple table lookups, with treecc checking the completeness of the table for us.

9 Conclusion and Future Directions

Treecc provides a new way to attack the complexity of compiler implementation by automating error-prone tasks. It performs a large amount of semantic analysis on the program to ensure that common problems are caught early in the development cycle.

Because treecc is based on an aspect-oriented foundation, it allows the programmer to separate out concerns and deal with them individually. Treecc puts the whole system back together in the most efficient manner possible.

The system is not necessarily complete. We'd like to experiment with rule-based code generation techniques. At present, optimizers and code generators must be written by hand, as operations on node types.

A rule-based system would make it easier to build clever optimizers as a set of pattern matching directives. Operations are already a special class of pattern matcher, but they don't have any back-tracking and retry capabilities.

Another area that treecc can be applied to is the construction of "Just In Time" compilers. The first phase of the JIT process is the reconstruction of the intermediate code form of the program. This intermediate code typically takes the form of abstract syntax trees or three-address statements.

Treecc is well-suited to the management of JIT intermediate forms. Register allocation, dynamic flow analysis, and machine-dependent code generation can be added as JIT aspects. We are currently exploring the use of treecc to assist in the construction of a JIT for Portable.NET

About the Author Rhys Weatherley is a member of the DotGNU Steering Committee, and the author of Portable.NET.

Rhys graduated from The University of Queensland, Brisbane, Australia, with an honours degree in Computer Science in 1990. Since then, he has worked in a number of positions at Australian universities and US companies.

He returned to Australia in late 1999 to pursue his own interests, including the founding of his Free Software company, Southern Storm Software, Pty Ltd.

His computer interests include programming language design and implementation, network computing, and user interfaces.

For our collecting translations of **"free software"** in the first issue, we have received two new translations:

"prosto programje" in **Slovenian**, offered by *Primoz Peterlin* <primoz.peterlin@biofiz.mf.uni-lj.si> on Jan 18, 2002;

"Programari LLiure" in Catalan, offered by *Xavi Drudis Ferran* <xdrudis@tinet.org> on Feb 19, 2002. Catalan is a romance language spoken by around ten million people.

Both of the languages have other different words to express the free in "free of charge".