A Modern Civic Framework

Floris den Heijer[*]
FNWI, University of Amsterdam
floris@denheijer.com

May 11, 2015

Abstract

Abstract, todo

[*]Dr. Clemens Grelck

Contents

1 Introduction And Motivation

1.1 Context

About the compiler course and the role of the framework.

1.2 Scope

Included aspects: Developer friendlyness, focus on teaching objectives vs language implementation

Excluded aspects: Augmentation of teaching objectives, change of VM backend

1.3 Research Goals

To find weaknesses in existing approach and language and establish a new framework overcoming these limitations.

1.4 Research Approach

Analysis of existing framework, consideration of language dependent and independent factors,

2 Frameworks

What constitutes a framework? Which requirements are relevant for a compiler framework? Establish scope of framework for compiler course, basic requirements, areas of assistance.

2.1 Framework Definition

See J2EE Paper

2.2 Advantages And Disadvantages

See J2EE Paper

2.3 Requirements For A Compiler Framework

Based on course objectives, papers on compilers..?

3 Analysis Of Existing Framework

Analyzing or evaluating software architecture is not a trivial task, as demonstrated by (...). Key questions answered:

- Which aspects of the current framework arise from choice of language?
- Which aspects or methodologies result from architectural choices?

- What are the drawbacks and advantages of the chosen platform and architecture?
- Is the framework sufficient with regards to the set requirements?

3.1 Overview

This section will provide an overview of the architecture of the existing framework and it's relevant components. On a high level, the framework aims to provide a structured approach for phase-based transformations on a language agnostic AST definition. Key components of the framework are:

1. Abstract Syntax Tree definition
2. Code generation based for node access and traversals
3. Transformation pipeline

In addition it provides a documentation generator for the AST, a small base library for string manipulation, lookup tables and argument parsing and a testing suite. The framework is written in C and XLST and runs on most *nix distributions, provided they support the required packages [1].

The outline of the architecture is presented in figure 1. It makes heavy use of XSLT to transform abstract specifications for the AST, traversals and node composition into usable helper code. This code combined with the base library provides the macro's and header files required for the rest of the compiler. The Flex/Bison lexer/parser is automatically compiled and transformed into a usable phase. Next, user phases and additional code are compiled and traversal entry points are substituted in the phase specification file and included in the main compilation unit.

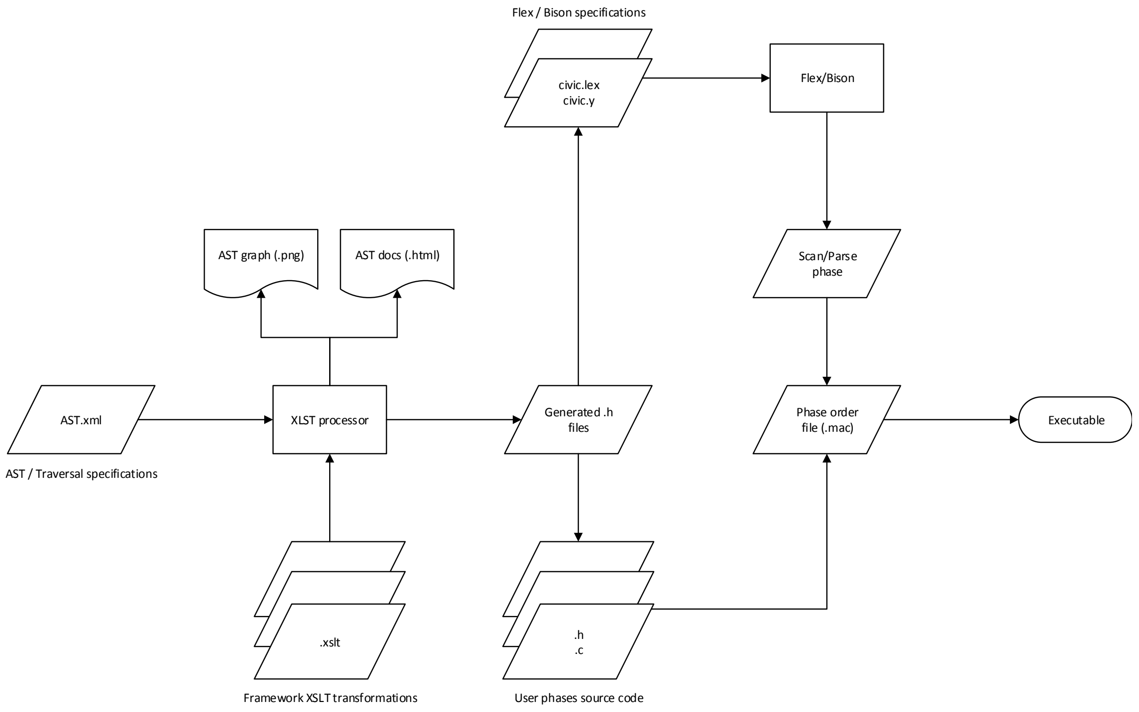

Figure 1: Simplified overview of the Civic framework

[1]gcc, gzip, flex, bison, xsltproc, dot, indent

3.1.1 Infrastructure

Concept of phase based transformation, concrete implementation dependent on AST.xml and phase.mac. Control over compilation process, debug helpers and base library. Exception handling.

3.1.2 Code Generation

Central in the approach taken by the framework is the seperation of specification and implementation. Specifications are given for:

- Nodes which form the abstract syntax tree, their attributes and child nodes
- Mapping from friendly attribute names to C-types, and relevant methods used for copying and free-ing
- Node constraints on:
  - Type: only certain child nodes or attribute are valid
  - Requirement: attributes or child node must be present
- Traversals, their mode of operation, affected nodes and function prefix

These specifications are transformed with XSLT to:

- Access macro's for nodes and their attributes
- Methods for initializing nodes
- Validity assertions
- User defined tree traversals and system traversals:
  - Check traversal
  - Free traversal
  - Copy traversal
- AST documentation (HTML) and graph (PNG)

3.1.3 Traversals & AST Modification

How is a tree represented (internally), and how to traverse?

3.1.4 Tree Validation & Type System

How is a tree validated, and how are type errors found?

3.2 Requirement Evaluation

Validate against set requirements.

3.3 Summary

Weaknesses:

- C language probable cause for:
  - Complicated traversal notion (due to macro's)
  - Check traversal could be made obsolete by including type checks in macro's
  - Free traversal (see GC)
  - Mixing of C-related implementation details in specification
  - Frequent use of debug helpers (doesn't help RAD)
  - Ancient debugging, verbose code, everything global, null pointers, etc.
- Platform probable cause for:
  - Extensibility issues (XSLT)
  - Architecture not obvious
  - Large specification required for small compiler (huge XML)
  - Documentation (due to perceived complexity)

Strong points:

- Everybody knows C (but nobody knows the framework)
- Platform is widely supported
- Forces rigid structure, deviation impossible due to non-transparency

4 Re-Designing The Framework

Key questions:

- Which programming languages are realistic alternatives to the current language, keeping in mind the current experience of students?
- Which areas or concepts could be improved or introduced to improve the framework's efficiency?
  - In what way are these concepts dependent on implementation language or on choice of architecture?
  - Which concepts are invariant of platform or architecture?

4.1 Language Considerations

As established in section 3.3 a large number of shortcomings can be attributed to the C programming language.

Which programming languages are realistic alternatives to the current language, keeping in mind the current experience of students?

Options: Java, Scala, C#, C++, Python, Functional (F#, OCaml)

Considerations: Student background (no functional, Java experience), New technologies accessible (ANTLR4, advanced debugging, exception handling), ease of use

Choice: C# (ANTLR, similarity to Java with more (functional) features, availability, cross-platform footnote, personal experience, managed language, extension methods)

Implications for platforms: Windows, Mono, base-CLR, etc.

4.2 Infrastructure

Code based configuration of phases, though overall similar approach.

4.2.1 Error handling

New helper methods.

4.3 Code Based & Type Safe AST

The current framework fails to completely seperate specification from implementation. But even an imaginary framework which succeeds has to interface with code at some point.

Considering code and AST are directly related, improvement is to construct AST from C# classes, offering type safety accross traversals, inheritance options and more extensibility when compared to XSLT generated code.

Honorable mentions: interface based approach, attribute tagging for order of child nodes, code generation based on inheritance graph (for visitors)

Language dependency: Any managed OOP language supports this construct, though C# additionally allows easy construction of object graph using bracket notion.

Platform dependency: Breaks with AST XML definition.

4.3.1 Collections

When given a strongly typed object graph, why not utilize the built-in support of list like structures? Discussion of problems.

Case with collections: - Node child may be node (or derived) or list of nodes - Discriminated union in C# (https://stackoverflow.com/questions/3151702/discriminated-union-in-c-sharp) - What doesn't work: Every node is also a list of nodes - Cyclic dependency in inheritance (Node : IList?) - Advantages: easy to add, iterate and replace - Disadvantages: Traversal issues (can a handler return a list?), complicates everything

Case without collections: - More natural to traversals - Singly-linked or doubly-linked - Difficult to append to or modify - Can be overcome with extension methods

Outcome: chosen for no collection types in lists, to avoid confusion with traversals.

4.4 Type-Safe Tree Traversals

AST transformation through tree traversals were and still are key components of a compiler framework. The original framework relied on configuration based code generation to create the necessary traversal paths and helper methods, resulting in indirect and cumbersome configuration of traversals.

AST transformation is crucial and diverse. Operations such as type checking, code generation, variable initialization checks, etc. Could be implemented as *TypeCheck()* on objects, but this doesn't scale. Type checking operations should be kept together, etc.

Extrinsic visitor: Easy addition of new methods, seperation of dissimilar operators, concrete elements visited is small, trade runtime performance for reduced intrusiveness of visitor pattern (Nordman)

Walkabout is exactly what I built (essense of visitor pattern).

When transforming an AST, it makes sense to use a type of depth-first traversal as actions on child nodes are often dependent on the root node and less on their cousins.

Requirements:

1.

The visitor pattern is a well established software pattern which separates algorithm from object structure. Specifically, it tackles traversal of a structure whose objects are part of an inheritance hierarchy.

In the traditional double-dispatch version this is done by adding an *accept(IVisitor)* method to all participating objects which directly return control to the visitor, invoking the *visit(Object)* method on the visitor determined by the run-time type (Essense of the visitor pattern).

There are many variations on the visitor pattern, such as the extrinsic visitor pattern (Nordman), reflection, type test and delegate (C# patterns), hierarchical etc (C2.com).

Introduction to composite and visitor pattern, example with no inheritance.

Review of existing hybrid visitor pattern, does not translate to with inheritance. Visitor may return more generic node than given. Invoked method depends on declared node type (single dispatch).

Double dispatch, calling visitor method with runtime type. However: - does not allow handling of types higher up in the inheritance chain - not natively supported by C#, even possible?

Alternatives (unsure): - store parent in every node, but who is responsible for setting it? Lifts responsibility to handler. Which property handled?

Introduce case with multiple matching methods, implies declaration order matters.

Define handler: - type in - type out - action or function (can be method invocation)

Define visitor: - ordered list of handlers - generic visitor method with reflection lookup

Added features: - type out omitted -¿ wrap with return of same type - conditionals on handler declaration

AbstractA / ConcreteB ConcreteC — ConcreteB2

Visitor with: AbstractA Visit(AbstractA node) ConcreteB Visit(ConcreteB node) AbstractA Visit(ConcreteB2 node)

Goal: preserve type safety in visitor methods without using casts: wrong: expr.Left = (Expression)Visit(expr.Left); good: expr.Left = Visit(expr.Left);

Suppose transformation of IntConst to Cast (both Expression) and default visit method: Expression Visit(IntConst node); T Visit(T node);

Assume we have a binop node with properties Left and Right of type Expression: var int1 = new IntConst(); var int2 = new IntConst(); var binop = new BinOp Left = int1, Right = int2 ;

Invoking the visitor like this results in invocation of the default Visit handler. binop.Left = Visit(binop.Left);

Lets follow the Walkabout pattern for the generic visitor method: T Visit(T node) where T : Node if (node != null) // If this visitor has a public method for this class, with convertible return type, invoke. var nodeType = node.GetType(); var method = this.GetType().GetMethod("Visit", new[] nodeType); if (method != null && (method.ReturnType == typeof(T) —— method.ReturnType.IsSubclassOf(typeof(T)))) return (T) method.Invoke(this, new [] node);

// Otherwise traverse children. VisitChildren(node);

return node

Suppose default visit children method which traverses and sets child node properties: void VisitChildren(Expression expr) foreach (prop of expr.GetNodeDerivativeProps()) Node value = (Node) prop.GetValue(); if (value != null) value = Visit(value); prop.SetValue(newValue);

GetValue() and SetValue operate on objects, the cast to Node is only required to ensure the most general case of T Visit(T node) functions.

Invoking the VisitChildren method on the BinOp successfully replaces both IntConst's with Casts.

Now suppose we want to write a method which acts on ALL expressions: Expression Visit(Expression expr);

We need to establish rules on precedence. The most intuitive approach would be to assume the most concrete visitor method should be called, similar to single-dispatch.

Now that we have handlers on two inheritance levels, invoking the most concrete handler involves a dynamic cast. binop.Left = Visit((dynamic) binop.Left); binop.Right = Visit(binop.Right);

VisitChildren also works because the GetMethod reflection call takes into account inheritance hierarchy.

But is this approach type safe? A dynamic object is essentially something of type 'object' and exists only at compile time (MSDN). The fundamental weakness in this approach lies in the premise that a node handler can return something more abstract than itself. To introduce a run-time error all we have to do is make the IntConst handler more abstract by returning a node: Node Visit(IntConst node) return new Node();

This runtime error is thrown because the property on BinOp requires an Expression and the generic visit method has not taken this into account. The value of a property is always returned as an object, and the property field type is retrieved using reflection. This means we cannot cast to the required type resulting in a perfectly working visitor returning an object too generic.

It is possible to supply this information to the Visit method without exposing it to outer classes. First the generic visitor and VisitChildren methods will be placed in an abstract class from which a new concrete visitor will inherit. Next, the visitor function is modified and made private: T Visit(T node, Type maxUpcast) where T : Node ... if (method != null && (method.ReturnType == maxUpcast —— method.ReturnType.IsSubclassOf(max

A wrapper method is introduced which hides this and restores the original behavior: T Visit(T node) where T : Node return Visit(node, typeof(T));

Finally the VisitChildren function is adjusted to pass along the type of the property: value = Visit(value, prop.PropertyType);

Instead of throwing a runtime error, the visitor now assumes the default case, skips invocation and jumps straight to visiting children. Static type checking is still performed by the C# compiler, preventing this from compiling: binop.Right = visitor.Visit((BoolConst)binop.Right);

The only case which is not supported in this scenario is one where there are two matching methods with differing return types: Expression Visit(IntConst i); IntConst Visit(IntConst i);

Drawbacks: always have to invoke visitor with dynamic cast when also targeting higher up the inheritance hierarchy.

4.4.1 Alternative approach: lambda based

While this approach is sufficient for most traversals, it requires some care when multiple inheritance layers are targeted. Method invocation depends on two separate mechanisms, namely direct (single-dispatch via compiler) and reflection based. This ambiguity can be avoided by preventing direct access to visitor methods. In this

Examples

4.5 Parser / Lexer Extensibility

More flexible Parser/Lexer generator support, such as ANTLR4 (LL(*) with EBNF).

Language dependency: Plenty of parsers exist for any language, first version of AST most likely constructed in code; change of language not essential.

Platform dependency: Depends on extensibility of parser generator, ANTLR4 has multiple ways of traversing a generated parse tree, YACC does not (embedding of code in parse phase restricts construction of AST)

4.6 Platform Updates

NuGet

4.7 Documentation Generator

Inheritance graph plotter, AST graph plotter

4.8 Summary

- Change to a managed OOP language relieves memory allocation issues, simplifies code based AST definition
  - Direct control over AST
  - Inherent type safety
  - More flexibility in choice of parser/lexer generator
  - Base library removes need for wrapper methods and guarantees documentation
- Code configured pipeline allows easier manipulation of phases
- Action based traversal formalizes and extends existing traversal process with conditionals and inheritance based actions

Also: schematic of entire system

5 Discussion

Key questions:

- How does the new framework differ from other compiler platforms?
- What other factors could improve the efficiency of the framework?

- vNext immutable AST -

6 Conclusion

7 References

References

Source? Everything explained vol. 2. March 2012.