A Modern Civic Compiler Framework

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

November 12, 2014

*Dr. Clemens Grelck

Contents

Language selection

Realistically the choice is one of C, C#, Java, F# or Scala, though functional languages are not prefered.

Considering a large portion of the project (so far) is spend on prototyping, the author's familiarity with C# and the fact that C# is syntactically similar to Java, the rest of the draft will feature this language. It must be noted that C# supports constructs such as lambda expressions and co- and contra-varianct interfaces, which Java does not.

This is a draft, this section will be supplemented with proper arguments.

Other topics

- AST vs ASG
  - Traversals annotate and augment AST to create some form of graph
  - Current framework explicitly states which nodes are to be traversed as 'sons', this means not all properties of a class of type 'Node' are to be traversed
- Is an AST more than a parse tree validator?
- Related patterns: Composite, Visitor, MultiMethod, Hierarchical Visitor Pattern, Graph traversals
- Continuation Passing Style
- Paper: On understanding types, data abstraction, and polymorphism (1985)
- It would be nice to have an aggregate visitor or a traversal which can be invoked on a (sub-)tree.
  - Following the Command Query Seperation principle, this traversal should not modify the graph

Composite and visitor design pattern

1 Analysis of the current framework

Short introduction on the framework, functionality outline, boundaries, etc. It makes sense to begin with a review of the AST.

- Structured source code transformation with phases
- Abstract syntax tree defined seperate from code
- Code generator for AST nodes, traversals, validation phase
- Debug helpers
- Documentation generator for AST
- Wrappers for common C functionality and extensions

1.1 Abstract syntax tree

The abstract syntax tree - or AST - is the heart of the framework. It defines the relation between nodes and is able to validate a tree and possibly prohibit invalid operations. It is defined in an XML file, which is parsed on compilation by a code generator. This generator in turn creates C code for the node structures, property accessors, construction helpers, traversals and validation mechanisms.

The generated node struct is a tagged union, which uses a tag field to identify the type of the node. C does not provide compiler guarantees on tagged unions, though limited type safety can be enforced by strictly controlling access to node instances [ref]. The framework does this by generating property accessor typedefs which are the only way to modify or access node attributes. In these typedefs debug assertions are placed which guarantee the attribute accessed belongs to the node.

Additional insurance is provided by the traversal function, which only invokes traversal handlers if the node type matches. [does it guarantee this?] [Todo: can user code set the node type? If not, who can?]

An AST node may require it's children to be of a specific set, for instance a 'while-loop' has a child 'condition' which is an expression. An expression could be a variable reference, function call or ternary, but not a statement. This construct is supported by introducing 'node sets' in the AST, which are a list of node types. Node declarations may reference these sets within the AST to constrain attributes. [Todo: are node sets also checked when updating a child variable? Is there logic in the typedef or in the validation phase?]

The AST definition is more than just a specification, as it is responsible for the correct translation of:

- Node types for the code generator, as well as the C types of attributes
- Traversal phases and targets
- Additional attribute validation

1.2 Traversals

1.3 Error handling

1.4 Strength weakness analysis?

Strength-weakness analysis without mud throwing and without AST, as this is handled below.

This analysis demonstrated that reasonable checks are in place to prevent basic type mix-ups. While this layer of protection is enough for experienced framework programmers, it is in no way a replacement for a modern type system.

2 Moving to a managed language

One of the objectives of this project is to investigate how we might implement an the framework in language which supports classes, polymorphism and better compile time type safety. The lack of a type system is likely one of the reasons for the seperation of specification and declaration, as guaranteeing strict access to union structs is tedious and easily done wrong.

When stripped of its type safety mechanisms the framework looks very different.

The strict access control to node attributes is handled by compiled time type safety, which raises the question if there is any benefit in seperating specification from code. A hand written AST definition is arguably more error prone than code, as code has the benefit of compile time checks which XML does not.

The node sets could probably be replaced in its entirity with class inheritance. User defined functions are no longer necessarily global, but are instead compartmentized in namespaces. Wrappers for C system functions are replaced with a managed framework, offering better documentation and greater ease of use.

In a language with type safety the framework - as is - does not make much sense.

2.1 Repositioning the AST

An interesting thought is to position the AST firmly in the user domain. In light of the old framework this is freedom of preposterous proportions. Is this always a good thing?

When defining an AST in pure code, it is important to maintain a clear picture of what is reasonable and what is not. If the subject matter is not known it's not very difficult to invent esoteric solutions to problems which never would have existed before.

One way to regain territory is by compensating in other areas. If the built-in visitors do not support the AST, most likely a student would change his code. But then again, in a decoupled system he or she could also replace whatever functionality is not wanted and soldier on. In a way this hypothetical scenario accomplishes what the original framework couldn't do: promote interaction with the underlying system. The only trouble is that so far we don't have anything resembling a framework yet.

2.2 Design considerations

In this section the influence of the structure of an AST with regards to user- and framework interation is examined. What is the best way to handle collections and how does

this effect traversal? Can we identify all reasonable implementations for a code based AST and how does any of this effect type safety?

2.2.1 Representing collections

Collections are used to represent a node which has a variable number of child nodes. There are at least two ways to represent such a collection in a tree:

1. Linked lists embodied in the tree structure (figure 1 and 2)
2. List or dynamic array as an attribute of a node (figure 3)

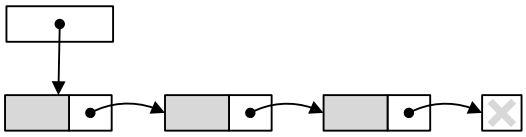

Figure 1: Singly linked list

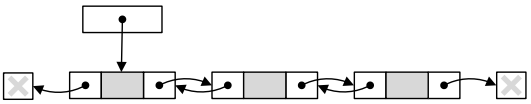

Figure 2: Doubly linked list

Figure 3: Dynamic array or list structure

Child nodes (shown in grey) are in held in a wrapper object responsible for pointers to other wrappers in the case of a linked list, or the maintaince of the list itself. While both constructs are similar in that expose an ordered set of child nodes, they are not interchangeable.

Wrappers for linked lists can be implemented as nodes with navigational properties, Additionally

Deep copy

The original Civic framework uses code generation to produce a traversal which copies (a portion of) a tree. In C# this requires a deep copy of the root object, which is not supported by default. There are several ways to create a deep copy:

1. Add clone() method to base class; each node must implement this call (fragile, manual labour)
2. Serialize object, then deserialize (might skip private fields)
3. Use reflection recursively, extend object (incomprehensible)

It just so happens the last method is the most effective, and supported by a library (under MIT license). Source code is available. It would be foolish to implement this manually, mistakes are easy to make and it would clutter the framework with more reflection code.

Phase validation

An AST can be invalid in a number of ways [also, is the AST the template, or definition? or the parsed tree?] For instance, it might contain a node of a wrong type in a certain position. These errors are mostly eliminated with subclassing. Another type of invalidity might come from the lack of a node in a position.

Variables of an object type in languages like C#, C++ or Java may have a null value; null can be assigned in any position where one might expect an instance of such an object.

Ideally we would like to perform certain validations at certain points in the transformation of the AST. When using backlinks created in an earlier phase it would be useful for future phases to skip assertions on these links.

Dynamically determined, validation traversal? Bound to phase or order of phase. Postcondition.

Only useful for debugging? Null reference isn't too bad during development.. Concrete examples needed

Code contracts in C#

Precondition, postcondition, invariant, result.

Scaling

How does any approach scale to larger compilers? What internal structures do they use? (AST transformation through phases, seperate AST for each phase).

Variable initialization scope

Needs to be clarified, two different styles methods exist: needs clarification.

```
1  int x = 1;
2
3  void proc() {
4    //  x from global scope or self reference?
5    int x = x;
6
7    void proc() {
8      //  also for local funcs
9      int x = x;
10   }
11
12   for (int x = x, x, x) {
13     //  .. and iterators
14   }
15 }
```

Listing 1: Ambiguity in variable references