

BSc. COMPUTER SCIENCE



UNIVERSITY OF AMSTERDAM

Modernizing the Civic compiler framework

Floris den Heijer
5873355

June 15, 2015

Supervisor(s): dr. C. Grelck

Signed:

Abstract

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetur. Fusce non massa eget quam luctus posuere. Aenean vulputate velit. Quisque et dolor. Donec ipsum tortor, rutrum quis, mollis eu, mollis a, pede. Donec nulla. Duis molestie. Duis lobortis commodo purus. Pellentesque vel quam. Ut congue congue risus. Sed ligula. Aenean dictum pede vitae felis. Donec sit amet nibh. Maecenas eu orci. Quisque gravida quam sed massa.

Contents

1	Introduction	5
1.1	Problem Statement	5
1.2	Context	5
1.3	Objectives	5
1.4	Research Approach	6
1.5	Scope And Limitations	6
2	Analysis Of Civic Framework	7
2.1	Architecture	7
2.2	Code Generation From XML	8
2.3	Flexible Traversals	10
2.4	Tree Validation	10
2.5	Development Environment	11
3	Re-design: Code Over Specification	13
3.1	Language Selection	13
3.2	Type-Safe AST	14
3.3	Improved Traversal Mechanism	16
3.3.1	View-Only Visitor	16
3.3.2	General Purpose Visitor	18
3.3.3	Lambda Visitor	19
3.4	Pluggable Architecture	20
3.5	Collections	21
3.6	Development Environment	21
3.6.1	Updated Lexer/Parser	22
3.6.2	Platform Updates	22
3.6.3	Documentation Generator	22
3.7	Validation	22
4	Discussion	23
5	References	25

Introduction

1.1 Problem Statement

Computer Science undergraduates at the University of Amsterdam have the option to learn the essentials of compiler construction by taking an eight week course where the main deliverable is a working compiler for a C-like educational language dubbed Civic. Construction of the compiler is assisted by a toolchain which consists of a custom virtual machine, assembler and development framework. In general, the course is received well and produces functional compilers. A recurrent issue however is the difficulty in getting up to speed with development, largely due to the technical challenges the chosen language (C99) poses and the lack of documentation for the framework.

This paper will present an analysis of the compiler framework and the concepts on which it is based. A new framework is proposed which adapts these concepts and implements them in C#, a modern managed language.

1.2 Context

Purpose of framework, teaching objectives, timeline. Other compiler frameworks, LLVM as backend, research on teaching methods for compiler construction??

The Civic framework is used in a well-defined educational setting and as such omits many aspects of more advanced, general purpose compiler frameworks. Teaching objectives dictate the primary focus to be on the development of a fully functional frontend (scanning, parsing, semantic analysis), rather than a backend suited for heavy optimization.

TODO

1.3 Objectives

The primary goal of this project is to analyze what a transition to an object-oriented language affects the Civic compiler framework. Key questions:

- Which concepts make the foundation of the current framework?
- Are these concepts implemented differently in a modern OO-language?
- What benefits might an OO-language add to the framework?

TODO

1.4 Research Approach

Analysis of existing framework, consideration of language dependent and independent factors, limitations of analysis.

TODO

1.5 Scope And Limitations

Focus is placed on creating a flexible abstract syntax tree which is suitable for basic optimization and code emission, eliminating the need for a separate intermediate representation. While greatly simplifying the compiler pipeline, this reduces the framework to a near 'frontend-only' role and limits comparisons with other frameworks.

This paper is limited to the compiler framework of the toolchain. This paper will not cover analysis of the Civic VM, assembler, teaching objectives or the suitability of alternative target languages such as LVVM, CIL or Java bytecode [3][9]. Analysis of the existing framework is limited to the extraction of key concepts.

Analysis Of Civic Framework

This chapter will provide an overview of the existing framework and its conceptual foundation. On a high level the framework aims to provide a structured approach for phase-based transformations on a language agnostic AST. Central to the framework is the specification of the abstract syntax tree (AST) and a code generator which creates macro's to access these nodes from code, as well as traversal entry points and control mechanisms. Automatic traversals which copy, free or check a (sub-)tree are also generated.

The Flex scanner generator and Bison parser generator is integrated by default and a basic configuration is provided which allows parsing of a subset of the Civic expression language. In addition it generates documentation on the AST, provides a small base library for string manipulation, lookup tables and argument parsing and a testing suite.

2.1 Architecture

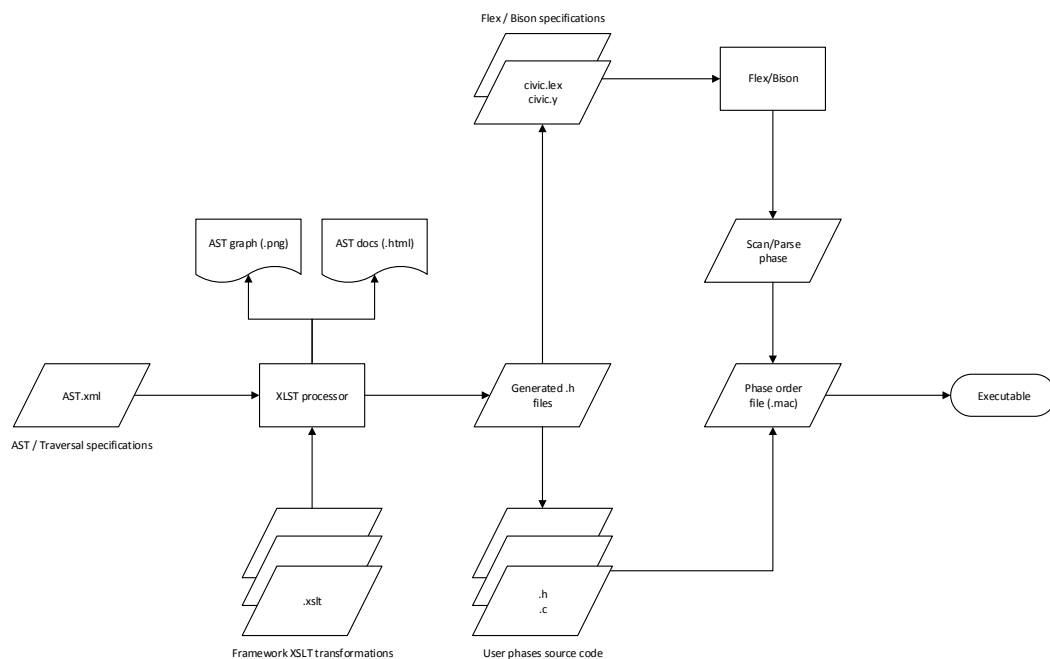


Figure 2.1: Schematic overview of the Civic framework

The framework is written in C and XLST and runs on most *nix distributions, provided they support the required packages ¹. A schematic overview of the architecture is presented in figure 2.1. The framework relies on an XSLT processor to transform an AST and traversal specification into framework code used by the rest of the compiler. This process is triggered on each build and replaces key framework header files once the specifications are altered.

User code is largely concentrated in AST transformations known as sub-phases, examples of which are type checking and context analysis. These sub-phases are grouped into logical phases, such as semantic analysis or code generation. Phases play a role in AST validation as described in section 2.4 and are integrated into the command line, providing the option to only compile up to a certain phase, or output extra debug information during a phase. During the compilation of the compiler, phases are flattened into a sequence of traversals and control code is added which pipes the output of one traversal into the next.

Names for library functions and phases adhere to an uppercase prefix followed by a lower camel-case identifier ([PFX]someId).

2.2 Code Generation From XML

One defining characteristic of the framework is an abstraction layer over the AST and traversal mechanisms. To better understand this layer, this section will first summarize how nodes and traversals are internally represented, then detail how the abstraction assists in this process and conclude with a summation of all parts of the framework affected by the layer.

Nodes are stored as a C-struct containing the type of the node and two structures representing their children and attributes (snippet 2.1). For each node type there are two data structures with it's specific attributes and references to child nodes, the union of which is referred to in the NODE struct (snippet 2.2). Access macro's are created for each attribute and child node as well as (de-)allocation code.

Snippet 2.1: *Node structure*

```

1  struct NODE {
2      nodetype          nodetype;      /* type of node */
3      int                lineno;        /* line of definition */
4      node*              error;         /* error node */
5      struct SONUNION     sons;         /* child node structure */
6      struct ATTRIBUNION  attribs;     /* attributes node structure */
7  };
8
9  typedef struct NODE node;
```

Snippet 2.2: *Sample child node and attribute union structures for node*

```

1  struct SONS_N_VARDEC {
2      node *Init;
3  };
4
5  struct ATTRIBS_N_VARDEC {
6      char *Id;
7      char *Type;
8  };
9
10 struct SONUNION {
11     struct SONS_N_VARDEC *N_vardec;
12     ..
13 };
14
15 struct ATTRIBUNION {
```

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

```

16     struct ATTRIBS_N_VARDEC *N_vardec;
17     ..
18 };

```

Traversals do not have a well defined single structure but translate into control flow statements and traverse tables. Users can push a traversal onto the stack and request the start of a traversal (snippet 2.3), triggering a lookup to a table for the active traversal holding the function which should be called index on node type. By default all child nodes are traversed using a switch statement based on node type (snippet 2.4). The `INFO` struct passed along provides statefulness to receiving functions, though this is not essential for traversal.

Snippet 2.3: *Starting a traversal*

```

1  node *n = ..;      /* retrieved elsewhere */
2  INFO *info = ..;   /* retrieved elsewhere */
3
4  TRAVpush(TR_mytrav);
5  n = TRAVdo(n, info);
6  TRAVpop();

```

Snippet 2.4: *Excerpt from catch-all switch construct*

```

1  node *TRAVsons (node *arg_node, info *arg_info)
2  {
3      switch (NODE_TYPE (arg_node)) {
4          case N_program:
5              TRAV (PROGRAM_DECLARATIONS (arg_node), arg_info);
6              break;
7
8          case N_declarations:
9              TRAV (DECLARATIONS_DECLARATION (arg_node), arg_info);
10             ...
11     }

```

To illustrate the complexity involved, consider the introduction of a new node to the AST. To achieve this, every traverse table must be supplemented, the default traverse function must be updated, two new structures must be added to the node union, allocation and de-allocation routines must be written as well as access macro's and the copy, free and check system traversals must be modified. Besides being tedious, performing these tasks manually introduces a significant risk of error. The framework provides XSLT transformations of a specification to automate most of this process. It takes only a couple of lines of XML to add a node to the AST, and all previously mentioned tasks are automatically executed. This does however mean that the specification forming the abstraction is not isolated from the target language (or leaky). For instance, to define a node with a `char*` attribute requires the specification to know about the difference in copying C-strings and integers.

Specifications and accompanying translations are given for:

- Nodes which form the abstract syntax tree, along with their applicable 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
 - Applicable phase: some attributes are not always required
- Traversals, their mode of operation, affected nodes and function prefix

2.3 Flexible Traversals

Internals of the traversal have been briefly shown in section 2.2 and will be explained in more detail in this section.

Traversals are supported in two modes: 1) user mode and 2) child mode. User mode traversals are required to implement handlers for every node type defined in the AST, child mode requires only handlers for a specified list of nodes. Every node which is not on the user specified list is routed instead to a catch-all function which automatically traverses child nodes of the unhandled node. This function is automatically updated when a node is added or removed or a change is made to any of the child node properties.

All traversals are elements in the <phases> element of the specification document, and must contain a unique identifier, readable short name, operation mode (user or child), associated header file and - depending on the chosen mode - a list of nodes handled by the traversal. When the compiler is built, enumerations of all node and traversal identifiers are generated. All nodes and traversals are assigned a zero-indexed integer to be used as an index. Every traversal adds an array of function pointers to the traverse table. The row is added at the index dictated by the enumeration and the array is indexed by the node index. References to the functions are constructed by concatenating the traversal identifier with the friendly name of the node. In child mode, unhandled node types are routed to the catch-all function explained in the last paragraph.

Every handler has the same function signature, requiring pointers to a node and an info structure, and returning a pointer to a node which replaces the current node. If a traversal is to be used as an entry point for a sub-phase, it must be added to the `phase.mac` file, where an entry point is referenced with the same signature as a handler function:

```
node *PFXnodeType (node *arg_node, info *arg_info)
```

The framework maintains a stack of active traversals manipulated with the `TRAVpush (id)` and `TRAVpop ()` functions. An active traversal can be controlled with the `TRAVdo (node*, info*)`, `TRAVopt (node*, info*)` and `TRAVcont (node*, info*)` functions. All return the result of the traversal of a given subtree, additionally the latter two respectively ignore null node arguments and invoke the catch-all handler.

This approach to traversals encourages users to create small, reusable traversals in similar way the visitor pattern is implemented in an object-oriented language [1].

2.4 Tree Validation

As explained in section 2.2, nodes all share the same structure. Semantically all node structures are valid trees, yet most combinations of AST nodes should yield an invalid tree. This section briefly explains the validation mechanisms present in the framework.

The first line of defense against illegal node operations is the tagged union implemented for nodes. A tagged union is a structure which can take on different but fixed types [12]. An `enum` is used to distinguish all different node types and used in the generated traversals to always pick the correct structure when traversing. Unlike functional languages, C provides no compile time checks to enforce the tagged union except for a simple check on the node type. However, access to nodes and traversal mechanisms are largely controlled from generated macro's which do provide a level of security. It must be noted that the macro property accessors do not check the node type, as they are used on both the left- and righthand side of assignments and assignments to conditionals are illegal in C.

The main validation mechanism lies in the 'check' traversal generated from the AST. This traversal walks the AST and ensures that every node has child nodes and attributes of the correct type, and that any attribute or child node marked as mandatory for the current phase is present. The check traversal must be manually added as part of a phase, so invalid trees are still possible during execution of any traversal up to the check.

2.5 Development Environment

Todo:

- Error handling
- Pre-configured scanner, parser
- Debug helpers
- Compact base library

Re-design: Code Over Specification

This chapter approaches the framework described in the previous chapter from the perspective of a modern object-oriented language.

Modern language, infrastructure approach, type-safety, code-based AST. 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?

In theory the existing architecture could be re-used with the code generator outputting C# instead, cleaner strategies exist which will be discussed in this section. The applicability of an abstraction layer will be tested by developing a model for a type-safe AST, followed by a proposal for a code-configured pluggable architecture.

Fundamentally changed: - language choice

- Code-First vs specification (for AST, traversals and phases - also validation omitted)
- Managed Language
- Package updates

Untouched or adapted:

- Traversal Mechanism (added lambda)
- Collection representation
- Base lib (arguments, default ANTLR4, documentation)

3.1 Language Selection

While C is arguably as modern and widely used as any other general purpose programming language, and can even be called object-oriented [11], it is definitely not strongly typed. Programming in C requires defensive programming and dynamic type casts to achieve a level of safety other languages get for free, as internally every structure is a `void*`. In this section the language options for a modernized Civic framework are briefly discussed.

Choosing a language is a non-trivial task and depends on many factors. For this project, the first criteria was that the language must be high-level and preferably managed to reduce the time spent

debugging memory allocation and null references in C. Student familiarity with the language was also considered, as well as the exclusion of any fully functional languages. The reason for the latter being removal of the functional programming course from the undergraduate program. One final consideration was the availability of relevant libraries and support infrastructure.

With functional languages excluded, procedural and scripting languages remain. A recent study found the most common languages on GitHub in these categories to be C++, C#, Objective-C, Java, Go, JavaScript, Python, Perl, Php and Ruby [8]. A study by Meyerovich and Rabkin found that 97% of computer science majors knew at least one imperative/OO language against 78% for dynamic languages [4]. Studies on static versus dynamic typing are somewhat inconclusive, though evidence suggests static typing improves maintainability, increases understanding of undocumented code and measurably reduces defects [2, 8].

When considering only statically typed managed languages, only C#, Java and Go remain. The TIOBE index is a programming language popularity index based on the number of hits in various search engines. While Java and C# occupy position 3 and 6 respectively, Go takes the 30th place, just below Ada [5]. With both C# and Java able to use the LL(*) ANTLR4 parser generator, either of these languages would suit the project.

Todo:

- C# has more powerful semantics, lambdas, properties, optional dynamic typing
- But it is somewhat platform bound, whereas Java is not

3.2 Type-Safe AST

Working with a strongly typed OO-language grants the ability to model AST nodes in greater detail than before. Different node types can be represented by different classes with node specific attributes exposed as properties, sharing a common interface or base class which can be used in traversals. Child nodes on a node should also be strongly typed and accessible in the same way as attributes. This poses an interesting challenge: if traversible properties are placed on the derived class, how can a traversal operating on the base class access them?

Two possible solutions come to mind: the first one - used in the existing framework - is to supplement the traversal with knowledge of the traversible properties, resulting in large switch statements. The second option is to mark traversible properties with an attribute (in Java attributes are known as annotations) and use reflection to retrieve them at runtime. A basic working version is shown in snippet 3.1. A note for those unfamiliar with C#, the { `get`; `set`; } represent an automatic property, similar to simple field-backed `get/set` methods in Java. The `.Where(p => p...)` method call on line 23 is an example of a lambda filter applied to all elements returned from `GetType().GetProperties()`, the `.Any(att => att...)` clause inside the lambda returns a boolean if any of the attributes declared on a property match the nested expression.

Snippet 3.1: *Using reflection to retrieve marked properties from a derived class*

```
1 class ConcreteNode : Node
2 {
3     [Child] public OtherNode Child1 { get; set; }
4     [Child] public OtherNode Child2 { get; set; }
5
6     public string Attribute { get; set; }
7 }
8
9 abstract class Node
10 {
11     // Cached list of traversible properties
12     private List<PropertyInfo> _properties;
13
14     // Returns a list of traversible children.
15     public IReadOnlyList<PropertyInfo> ChildProperties
```



```

16     {
17         get
18         {
19             if (_properties == null)
20             {
21                 _properties = GetType()
22                     .GetProperties()
23                     .Where(p => p.CustomAttributes.Any(att => att.AttributeType ==
24                         typeof(ChildAttribute)))
25                     .ToList();
26             }
27             return _properties;
28         }
29     }
30 }
31
32 public sealed class ChildAttribute : Attribute
33 {
34     // Attribute used only to mark properties
35 }

```

This basic setup fails on several counts. First of all, the order in which properties are declared on a class is not guaranteed to match the order in which they are returned from reflection. Second, it is possible to place the [Child] attribute on any attribute which is then added to the list of traversible children. C# 4.5 exposes declaration information via attributes, which is used to populate a field on the attribute. This order can be used to sort the list of properties, as shown in snippet 3.2.

Snippet 3.2: *Updated code for the base class and attribute*

```

1  abstract class Node
2  {
3      ..
4
5      // Returns a list of traversible children.
6      public IReadOnlyList<PropertyInfo> ChildProperties
7      {
8          get
9          {
10             if (_properties == null)
11             {
12                 _properties = GetType()
13                     .GetProperties()
14                     .Where(p.PropertyType == typeof(Node) ||
15                         p.PropertyType.IsSubclassOf(typeof(Node)))
16                     .Where(p => p.CustomAttributes.Any(att => att.AttributeType ==
17                         typeof(ChildAttribute)))
18                     .OrderBy(p => p.GetCustomAttribute<ChildAttribute>().Order)
19                     .ToList();
20             }
21             return _properties;
22         }
23     }
24
25     public sealed class ChildAttribute : Attribute
26     {
27         private readonly int _order;
28
29         // C# 4.5 only, see Microsoft.Bcl package for < 4.5
30         public ChildAttribute([CallerLineNumber] int order = 0)
31         {
32             _order = order;
33         }
34
35         public int Order

```

```
36    {  
37        get { return _order; }  
38    }  
39 }
```

The `Node` abstract base class and the `ChildAttribute` attribute form the basis of the approach chosen for a type-safe AST. Nodesets from the previous framework can be exchanged for (abstract) base-classes for easy targeting of expressions, statements, or variable declarations. Because C# (and Java) only support single-inheritance, a node is limited to only one branch of inheritance, which is probably a good thing.¹

It must be noted that the order in which child properties are returned is somewhat ambiguous when working with an inheritance hierarchy where traversible properties are defined on multiple classes. Properties are correctly retrieved, but are sorted on line number, which is valid only for a single file. This poses a problem for classes defined over multiple files. This behavior is arguably undefined, though such a list might be ordered first by order of inheritance, and then by line number. Sorting this way is non-trivial and unnecessarily costly, considering there were no real use cases for such hierarchies, so no action was required.

3.3 Improved Traversal Mechanism

This section reviews common traversal and manipulation mechanisms applicable for trees with nodes in an inheritance tree and presents implementations for the type-safe AST constructed in section 3.2. Also presented is a new adaptation called the lambda visitor, which is benchmarked against other visitors.

In the analysis of the existing framework the traversal mechanism was identified as some form of visitor pattern. The visitor pattern was first formally defined by Gamma et al. in 1994, who defined its intent as: “Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”. Parallels with the requirements for the traversal mechanism on an AST are easily drawn, coincidentally the one example they present is *about a fictional AST!* To quickly summarize the example, instead of separating compilation phases into logical units, functionality is placed in a method accessible on *every* node. Thus, the type checking phase would add an overridable `TypeCheck()` function to the base class, and responsibility for traversing child nodes is placed on the node as well.

The visitor pattern disapproves the modification of a node class for every new operation on a tree. It defines a `NodeVisitor` base class which contains a visit method for every node in the hierarchy (`VisitAssignment(AssignmentNode)`, `VisitDeclaration(Declaration)`, etc). A method accepting the visitor class is added to every node (`Accept(NodeVisitor)`), whose sole purpose is to return control to the visitor class by calling the visit method for that node. Each new operation of the tree results in a derived visitor containing only the relevant functionality.

The classic example of the pattern is not shown here, as it only demonstrates unwanted code. The main design goals for the upcoming three visitor implementations are a maximum reduction of clutter and a transparent algorithm. Less clutter means faster or at least more time spent on actual development and should everything go wrong, it should at least be clear where it goes wrong.

3.3.1 View-Only Visitor

The first algorithm presented traverses an AST as defined in section 3.2 and invokes publicly defined `Visit(...)` methods on the class. This scenario most closely resembles the default visitor

¹An effort has been made to implement an object composition model, which required too much documentation to explain and provided identical results.

pattern adapted for reflection. As anticipated in the design of the AST, a reflection based visitor requires meta-information on which properties are traversible and in what order. Access to child properties is provided by the `Node.ChildProperties` property, which returns a read-only list of child properties. Traversal of children is performed by the `visitChildren()` method, which closely resembles a reflection based version of the catch-all function in the C-based framework.

The view-only visitor is presented in snippet 3.4. Unbeknownst to the author of this paper, a generalized version of the visitor pattern titled *Walkabout* was presented in 2007 by Palsberg and Jay. The algorithm is (unsurprisingly) identical in form, and may be easier to follow if unfamiliar with C# (see snippet 3.3).

Snippet 3.3: *Walkabout algorithm by Palsberg and Jay*

```
1  class Walkabout
2  {
3      void visit(Object v)
4      {
5          if ( v != null )
6              if (/* this has a public visit method for the class of v */)
7                  this.visit(v);
8              else
9                  for (each field f of v)
10                     this.visit(v.f);
11      }
12 }
```

Snippet 3.4: *View-only visitor*

```
1  public abstract class ViewOnlyVisitor
2  {
3      public void Visit<T>(T node) where T : Node
4      {
5          if (node == null)
6              return;
7
8          // If the visitor has a public visit method for this object, invoke.
9          var method = GetType().GetMethod("Visit", new[] {node.GetType()});
10         if (method != null && method.ReturnType == typeof(void))
11         {
12             method.Invoke(this, new object[] {node});
13         }
14         // Otherwise, traverse children.
15         else
16         {
17             VisitChildren(node);
18         }
19     }
20
21     public void VisitChildren(Node node)
22     {
23         if (node == null)
24             return;
25
26         // Traverse child nodes using attributes and submit to visitor.
27         foreach (var prop in node.ChildProperties)
28         {
29             var value = (Node) prop.GetValue(node);
30             if (value != null)
31             {
32                 Visit(value);
33             }
34         }
35     }
36 }
```

3.3.2 General Purpose Visitor

Transformations often require the replacement of nodes with nodes of another type but with a common ancestor. In this section the view-only visitor is modified to support node replacement.

The flexibility found in the C-based traversal can be applied to the view-only visitor by adding return values to visitor handlers. It is important to realize that a transformation from type A to type B is not always applicable, even if they share a common ancestor. The reason for this is that every node in the tree is bounded by the type of the property. Thus, if a type A node is held in a property of type A, replacing it with a type B node would cause a runtime exception or compilation error and should be prevented.

An additional parameter is required on the generic visitor function which can be used internally by the `VisitChildren()` method to pass the enclosing type of the property. Consumers of the visitor should not need this type parameter, as they only invoke the generic visitor if no overload in their implementation matches, in which case the maximum upcast is as it was before, the generic type parameter. The general purpose visitor is presented in snippet ??.

This visitor is able to deal with out-of-bound handlers by utilizing two mechanics: 1) user code directly invoking the unfit handler won't compile because of static type checking and 2) automatic child traversal or forced invocations of the generic visitor are properly bound to a safe limit.

One final test remains, what happens when multiple matching handlers are defined? It is possible to write a handler for a base type as well as one for a concrete type. For automatic traversals, the most derived handler is automatically chosen because the runtime type is always used for method selection. However, user code is bound to the single-dispatch mechanism which selects a method based on the type of the *reference* holding the object. A derived type inside a variable of the base type will be routed to the base type handler.

This is not as bad as it seems as pre-compile inspection will show which method will be invoked. If this behavior is unwanted, the argument to the visitor function can be dynamically cast to delay method selection until runtime (since C# 4.0 [10]).

This leaves two unsupported cases:

1. Two or more handlers differing only in return type
2. Two or more handlers, where the derived handler returns a more generic type than the less derived handler

The first one is an illegal construct in C# and most other imperative languages which cannot be avoided. The second one is a result from reliance on the `Type.GetMethod(...)` function to choose a function based on the first argument. This matching process is done without accounting for suitable return type, so if the most derived guess fails the second test, no other options are evaluated. Solving this issue means a full implementation of multiple-dispatch, which is preferably avoided.

Snippet 3.5: Generalized visitor

```
1 public abstract class Visitor
2 {
3     public T Visit<T>(T node) where T : Node
4     {
5         return Visit(node, typeof(T));
6     }
7
8     private T Visit<T>(T node, Type maxUpcast) where T : Node
9     {
10        if (node != null)
11        {
12            // If this visitor has a public method for this class, with convertible
13            // return type, invoke.
14            var nodeType = node.GetType();
15            var method = this.GetType().GetMethod("Visit", new[] { nodeType });
```

```

15         if (method != null && (method.ReturnType == maxUpcast ||
16             method.ReturnType.IsSubclassOf(maxUpcast)))
17         {
18             return (T) method.Invoke(this, new [] {node});
19         }
20         // Otherwise traverse children.
21         else
22         {
23             VisitChildren(node);
24         }
25     }
26     return node;
27 }
28
29 void VisitChildren(Node node)
30 {
31     foreach (var prop of node.ChildProperties)
32     {
33         Node value = (Node) prop.GetValue();
34         if (value != null)
35         {
36             value = Visit(value, prop.PropertyType);
37             prop.SetValue(value);
38         }
39     }
40 }
41 }

```

3.3.3 Lambda Visitor

While the general purpose visitor is sufficient for most traversals, it requires some care when multiple layers of an inheritance hierarchy are targeted. Method invocation depends on two separate mechanisms (single-dispatch and reflection), one of which is not very flexible. As an alternative to a full multiple-dispatch implementation, this section will discuss the lambda visitor, which removes publicly defined method altogether. Performance benchmarks are also presented for the general purpose visitor, the lambda visitor and several optimizations of the lambda visitor.

Eliminating single-dispatch means replacing the publicly defined visitor handlers with an ordered list of delegates and configuration functions to modify the list. The basic traversal mechanism remains the same, but instead of probing for publicly exposed methods the list of handlers is consulted instead. Multiple-dispatch related ambiguity with targets across multiple layers of inheritance is solved by explicitly stating the order of evaluation. Such handlers must contain at least:

1. Type of the node handled
2. Return type of the handler
3. Delegate containing the transformation function

The distinction previously made between replace method and 'view-only' methods can be removed by offering two types of delegates. In .NET these are represented by the `Func<TArg, TRet>` and `Action<TArg>` delegates, indicating respectively a function which takes an object of type `TArg` and returns an object of type `TRet` and a function which takes an object of type `TArg` but doesn't return anything (void function). These can be interleaved as in returning nothing in the context of tree traversals indicates "no change" to the initial object; it would behave as if it would return itself.

The logic determining which handler should be invoked is now entirely encapsulated by the visitor, a simple but useful feature would be an optional predicate preceding invocation of the function. This, together with a lambda-style approach to defining visitor methods allows for more configuration flexibility. Such handlers must additionally hold:

4. Delegate type (void or returns)
5. Delegate containing a predicate

Additionally, fall-through handlers could be supported with this approach, though the decision not to do so is based on the fact that fall-through statements have been mostly expelled from switch constructs in modern languages as they are a frequent cause of bugs.

A documented version of the lambda visitor is presented in snippet 3.6. Several performance improvements could be made, as the `DynamicInvoke` function is supposedly notoriously slow [7].

Snippet 3.6: *Lambda-based visitor*

```
1 private T Visit<T>(T node, Type maxUpcast)
2     where T : Node
3 {
4     if (node == null)
5         return null;
6
7     // If a handler exist for this class with convertible return type, invoke.
8     var nodeType = node.GetType();
9     foreach (var handle in _handlers)
10    {
11        // Test for matching type (or subtype).
12        if (!handle.Type.IsAssignableFrom(nodeType))
13            continue;
14
15        // Test for fitness of transformation result.
16        if (!maxUpcast.IsAssignableFrom(handle.MaxUpcast))
17            continue;
18
19        // Test for predicate.
20        if (handle.Predicate != null)
21        {
22            bool result = (bool) handle.Predicate.DynamicInvoke(node);
23            if (!result)
24                continue;
25        }
26
27        // Check if this is an action or function.
28        if (handle.Function != null)
29        {
30            return (T) handle.Function.DynamicInvoke(node);
31        }
32        else
33        {
34            handle.Action.DynamicInvoke(node);
35            return node;
36        }
37    }
38
39    // If no handlers are found or if all fail, traverse children.
40    VisitChildren(node);
41
42    return node;
43 }
```

3.4 Pluggable Architecture

- XSLT not necessary for AST generation, or check/copy travers (present copy function)

Phases reduced to list of traversals, as there is no need for logical separation. - phases defined with interface, code-based configuration of phases (optionally based on console arguments) - grouping easy to add if required

3.5 Collections

While an AST as defined in section 3.2 has more depth than what was previously possible, it is still the same basic tree structure as before. One area which could be improved is in the representation of a collection of nodes. A function may have an arbitrary number of parameters, variables, nested functions and statements, is it possible to represent these structures more clearly with built-in lists?

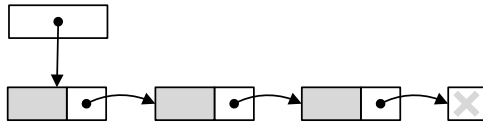


Figure 3.1: *Singly linked list*

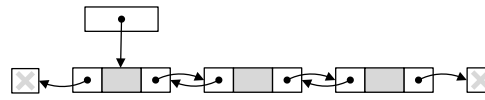


Figure 3.2: *Doubly linked list*



Figure 3.3: *Dynamic array or list structure*

A variable number of nodes is implemented in an AST by nesting container nodes up to the depth required, forming either a singly-linked list (fig. 3.1) or a doubly-linked list (fig. 3.2). A node holding any number of child nodes might be represented in C# as in snippet 3.7.

Snippet 3.7: *Example of list attributes in a function definition*

```
1 class FunctionDefinition : Node
2 {
3     [Child] public List<VarDec> Variables { get; set; }
4     [Child] public List<Statement> Statement { get; set; }
5     [Child] public Expression Return { get; set; }
6 }
```

This poses the question how such a list should be treated. I should we see a list of nodes as a special type of node with no traversible children but traversible or can the functionality `ChildAttribute` defined in section 3.2

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.

3.6 Development Environment

Implications of unmanaged vs managed, reduction of base library, debug upgrade..?

3.6.1 Updated Lexer/Parser

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)

3.6.2 Platform Updates

NuGet

3.6.3 Documentation Generator

Inheritance graph plotter, AST graph plotter

3.7 Validation

Type-safety is more than enough?

Old method was not type safe, simply prevented untracable errors

Tree corruption unnoticed until check routine ran

Classes as f

Discussion

vNext immutable AST,

Language choice, developers switch language often We found that developers rapidly and frequently learn languages. Factors such as age play a smaller role than suggested by media. In contrast, which languages developers learn is influenced by their education, and in particular, curriculum design.

Functional Promisingly, developers who learned a functional or math-oriented language in school are more than twice as likely to know one later than those who did not.

References

- [1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [2] Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Eric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(5):1335–1382, 2014. ISSN 1382-3256. doi: 10.1007/s10664-013-9289-1. URL <http://dx.doi.org/10.1007/s10664-013-9289-1>.
- [3] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2002.
- [4] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. *SIGPLAN Not.*, 48(10):1–18, October 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509515. URL <http://doi.acm.org/10.1145/2544173.2509515>.
- [5] S. Nanz and C. A. Furia. A Comparative Study of Programming Languages in Rosetta Code. *ArXiv e-prints*, August 2014.
- [6] Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC’98. Proceedings. The Twenty-Second Annual International*, pages 9–15. IEEE, 1998.
- [7] Joel Pobar. Dodge common performance pitfalls to craft speedy applications, 2005. URL <https://msdn.microsoft.com/en-us/magazine/cc163759.aspx>.
- [8] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635922. URL <http://doi.acm.org/10.1145/2635868.2635922>.
- [9] SAC-Home. Homepage of the sac project, May 2015. URL <http://www.sac-home.org/>.
- [10] Herbert Schildt. *C# 4.0: The complete reference*. McGraw-Hill, 2010.
- [11] Axel-Tobias Schreiner. *Object-Oriented Programming With ANSI-C*. Hanser.
- [12] Source? Everything explained vol. 2. March 2012.