BSc. Computer Science

University of Amsterdam

An object-oriented implementation of the CiviC compiler framework

Floris den Heijer
5873355

June 21, 2015

Supervisor(s): dr. C. Grelck

Signed:

Abstract

Aenean sem dolor, fermentum nec, gravida hendrerit, mattis eget, felis. Nullam non diam vitae mi lacinia consectetuer. 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

CHAPTER 1

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 model language called 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, LVVM 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

- In main objective: use of strongly typed object-oriented language
- Could the key concepts be better implemented in an OO-language?
- What benefits might a strongly typed OO-language add to the framework?
- Will this approach yield better usability for the user?
- How does it compare in terms of framework maintainability?

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 seperate 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.

CHAPTER 2

Analysis Of Civic Framework

This chapter will provide an in-depth review of the existing framework, detailing it's architecture, workflow and technical implementation. The analysis at the end of the chapter will inspect aspects such as maintainability and ease-of-use.

it's technical implementation and it's conceptual foundation. 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 are 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.

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 validation 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 camelcase identifier (`[PFX]someId`).

Mention all features, plus validation!

2.1 Architecture

On a high level the framework aims to provide a structured approach for phase-based transformations on a language agnostic AST or *abstract syntax tree*. Central to the framework is a specification of the AST and a code generator which produces interaction code and traversal control mechanisms. Interaction code here means an interface through which AST nodes can be created or modified, and composed into a tree. Manipulation of a tree is achieved through *traversals*, which target specific set of nodes and specify code to be executed for each node, much like the visitor pattern introduced in the next chapter. The framework recognizes distinct *phases* of compilation, such as semantic analysis or code generation, and provides a way to group related transformations and execute them sequentially.

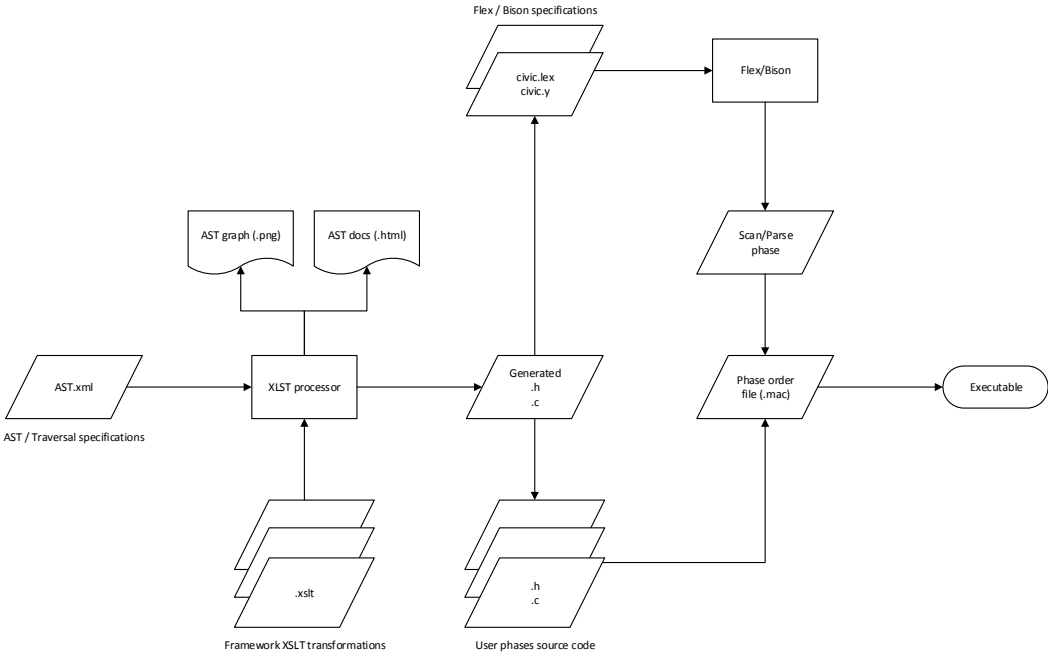

Figure 2.1: *Schematic overview of the Civic framework*

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 a specification of the AST and traverals into C header and source files which can be used by the rest of the compiler. This process is triggered on each build and replaces key files if the specifications are altered.

2.2 Workflow

The framework enforces a specific workflow and code organization structure. It presents the process of compilation as a pipeline of phases, each phase consisting of one or more traversals. Users can modify or expand the pipeline by introducing new phases and traversals, but are bound by this structured approach. Configuration of the pipeline is possible through code and supported by C macro's further explained in the next section. Seperated from code are the specification of the AST and traversal definitions.

This workflow directs user code largely to:

1. Specification of AST and traversals (`AST.xml`)
2. Traversal header and implementation files (user directory with `*.c, *.h` files)
3. Pipeline configuration file (`phase.mac`)

As each compiler requires scanning and parsing, a phase invoking the Flex scanner generator and Bison parser generator are provided by default. While strictly part of bootstrapping code, most users will work with these tools, adding to the above list:

4. Lex configuration for Flex (`civic.l`)
5. YACC configuration for Bison (`civic.y`)

Changes to the AST or the addition of a new traversal are propagated by building the compiler. The build process produces updated integration code as well as documentation on the AST in

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

two forms: 1) as an HTML document detailing every node and it's children, and 2) as an image depicting the AST graphically, with lines connecting nodes where applicable.

Debugging is assisted through command line arguments which can specify to only compile up to a certain point. In addition, the command line allows configuration of tracer options, which - assuming the user supplemented their code with trace statements - enable detailed output of the callstack.

2.3 Technical Implementation

The most defining characteristic of the framework is the abstraction layer over the AST and traversal mechanisms. To better understand this layer, this section will detail the internal representation of nodes, traversals and phases and discuss the role of code generation. *This section assumes knowledge of the C programming language.*

2.3.1 Abstract Syntax Tree

Listing 2.1: *Template for specification of a node*

```
<node name="[node_name]">
    <sons>
        <son name="[child]">
            <targets>
                <target mandatory="[yes/no]">
                    <node name="[child_name]"/>
                    <phases><all /></phases>
                </target>
            </targets>
        </son>
    </sons>
    <attributes>
        <attribute name="[attr]">
            <type name="String">
                <targets>
                    <target mandatory="[yes/no]">
                        <phases><all /></phases>
                    </target>
                </targets>
            </type>
        </attribute>
    </attributes>
</node>
```

Listing 2.2: *Internal structure of a node*

```
struct NODE {
    nodetype            nodetype;       /* type of node */
    int                 lineno;         /* line of definition */
    node*               error;          /* error node */
    struct SONUNION     sons;           /* child node structure */
    struct ATTRIBUNION  attribs;        /* attributes node structure */
};

typedef struct NODE node;
```

Nodes in the AST are defined as a `<node>` XML element in the `<syntaxtree>` element of the `AST.xml` specification file. Every node has a name, and optionally a list of named child nodes (or *sons*) and attributes, following the template shown in listing 2.1. The difference between an attribute and a child node is that child nodes can be traversed, whereas attributes are simple properties, though they might in fact reference other nodes. The `<targets>` element inside either a child or attribute element is among other things used to declare a property as mandatory, this is further explained in section 2.3.4.

Internally, nodes are represented by the `NODE` structure shown in listing 2.2. This structure holds the type of the node as an `enum` and references to two union structures representing applicable child nodes and attributes. These structures are generated and union'ed based on the XML specification, which translates a `<son>` element to a `node*` member of the former and an `<attribute>` element to a typed member of the latter. The C-type used for members of the attribute structure comes from a mapping section in the specification.

Besides one being traversible, child nodes and attributes are accessed the same way through generated macro's. These macro's primarily exist to avoid manually selecting which union struct variable should be accessed. A side effect is that it also hides which properties are children and which are attributes, and it implies an attribute cannot have the same name as a child node. Access macro's *do not* ensure they are used on the correct node by validating the node type, as they are used both as a setter on the left-hand side and a getter on the right-hand side, and assignment to condidionals is illegal in C.

Creating nodes is a straightforward process as allocation code for each node is generated from the specification. Every node has a create function, which accepts references to child nodes as well as attributes marked mandatory. Free-ing nodes is a recursive operation which requires a traversal and is covered in the next section.

2.3.2 Traversals

Listing 2.3: *Template for specification of a traversal*

```
<traversal id="[PFX]" name="[Friendly Name]" default="[sons/user]" include=
 ↪  "[header.h]">
    <!-- TravUser element only applicable for 'sons' mode -->
    <travuser>
        <node name="[NodeName]"/>
    </travuser>
</traversal>
```

Traversals do not have a single, well-defined structure but rather translate into control flow statements and traverse tables. Every traversal must be declared in the `AST.xml` file as a child of the `<phases>` element and must adhere to the template presented in listing 2.3. They require a unique prefix, a friendly name, mode of operation and header file with handler declarations.

All handlers have the same function signature: passed in are pointers to an input node and info structure, a pointer to the output node is returned. The info structure may be defined by the traversal and can be used to share state between traversal methods. Handler declarations are not generated and must be declared by hand.

Two operation modes are supported: 1) user mode and 2) child mode. User mode traversals are required to implement handlers for every node type in the AST, child mode requires only handlers for a set of nodes specified in the `<travuser>` element. In child mode, every node not in that list is handled by a catch-all function which automatically traverses child until one a node is found for which a handler is registered.

To initiate a traversal, users must push it's identifier onto the stack and invoke the `TRAVdo` or `TRAVopt` traverse functions[2]. These functions have the same signature as handlers and should in fact be seen as a smart wrapper around them, choosing the correct handler based on node type and active traversal. Traversals may be nested by pushing another traversal onto the stack and calling the traverse function again, though not all traversals support this kind of initiation. More specifically, only traverals devoid of an info structure support this approach, as the allocation and de-allocation of this structure is always static to the owning traversal. Initiation of a traversal is illustrated in listing 2.4.

Once a node is passed into the main traversal function, a lookup on node type to the function table for the active traversal will return a handler for that node. The traverse tables are generated from specification, with user mode requiring function references for all node types, and child mode filling in the gaps with the `TRAVsons` function. This is the catch-all automatic traversal function which uses a large switch statement on node type to invoke the traversal function on every child node, the result of which is put back to the property. Once all children are traversed the input node is passed back to the original caller. An excerpt of a generated `TRAVsons` function is shown in listing 2.5.

Listing 2.4: *Example of traversal invocation*

```
{
    node *n, info *info;    /* retrieved elsewhere */

    TRAVpush(TR_mytrav);    /* push traversal ID */
    n = TRAVdo(n, info);    /* initiate */
    TRAVpop();              /* pop traversal */

    return n;               /* return control */
}
```

Listing 2.5: *Part of the generated TRAVsons catch-all function*

```
node *TRAVsons (node *arg_node, info *arg_info)
{
    switch (NODE_TYPE (arg_node)) {
        case N_program:
        TRAV (PROGRAM_DECLARATIONS (arg_node), arg_info);
        break;

        case N_declarations:
        TRAV (DECLARATIONS_DECLARATION (arg_node), arg_info);
        ...
    }

    return arg_node;
}
```

Three important traversals are generated on every build:

1. Copy traversal, which creates a deep copy of the structure (child nodes) and a shallow copy of all attributes
2. Free traversal, which de-allocates a tree and - if applicable - it's attributes
3. Check traversal, responsible for validating AST integrity

[2]TRAVdo does not allow traversal of a null argument, the TRAVopt function only invokes TRAVdo on a not-null argument.

Attributes are copied and freed based on their specification in the previously mentioned `<ast-types>` element, which identifies three situations: 1) node references, 2) strings and 3) value types and other references. Node references as attributes are not followed and only the reference is copied, freeing a node attribute only nulls the reference but leaves the structure intact. Strings are copied and freed using string library functions, and all other value and reference types are copied using straight C assignment. The check traversal will be discussed in section 2.3.4.

2.3.3 Phases

As briefly described in section 2.2, configuration of the phase pipeline is done through code with the assistance of helper macro's. Configuration is achieved through the `phase.mac` file, where users can define a phase as a logical distinction or practical grouping of traversals. A friendly name can be assigned to each phase, which, in debug mode, is written to `stdout`. In addition each phase is assigned a unique identifier which can be used for validation as detailed in the next section. Traversals in a phase must also specify a friendly name for the same debugging purpose as their parents and more importantly the name of the entry point to the traversal. Traversal entry points are nearly identical in signature to handlers, except they do not accept an info structure as an argument. Entry points usually allocate an info structure, though this may be ommitted, and push a traversal onto the stack, much like listing 2.4.

On compilation, macro's in the `phase.mac` file are resolved and the traversals are flattened into a sequence and wired up to feed the result of one traversal into the next. Debugging code which allows command line arguments to specify which phases should run is also integrated. At runtime the pipeline can be conditionally halted or aborted by the user during transformation with the `CTI*`[3] range of functions.

2.3.4 Tree Validation

As explained in section section 2.3.1 all nodes share the same structure, differing only on node type and which variables of two union structures are used. The nodetype `enum` in conjunction with the switching mechanism and access macro's provide the basis for a tagged union construct, though this is not enforced in any way. Thus, semantically all nodes are valid trees, yet most combinations of AST nodes should be recognized as an invalid tree.

This is where the check traversal mentioned in section 2.3.2 comes in. This generated phase validates all constraints set in the AST specification and aborts compilation on an invalid tree. Constraints can be set in the `<targets>` element optionally placed inside a child declaration or type element of an attribute (see listing 2.1). Both child node and attribute properties support the *mandatory* constraint, which invalidates the tree if found empty.

While a type definition is always required on an attribute, child nodes allow any node unless specified otherwise. The `<node name="..."/>` element signals the check traversals that there is only one node type valid for the child position. Often this is not sufficient, as children operate may operate on *sets* of nodes. An example of this is any expression language, where an operator may act on any expression, itself included. The framework allows flexibility in the specification of valid child nodes through *nodesets*, which are defined as children of the `<nodesets>` element. A nodeset has a name and list of named nodes which belong to the set. Nodesets may not be nested.

Constraints may only be valid during or after certain phases. The framework supports this by allowing multiple `<target>` elements for properties. Users can specify a range of phases to which the constraint applies, using the phase identifier from the last section. The example in listing 2.6 shows a hypothetical node which during early compilation phases may contain any node from the expression set, while in later phases the node must contain a reference to an integer constant node.

[3] CTIterminateCompilation, CTIwarn, CTIabort, CTIerror, CTIerrorContinued, etc.

Listing 2.6: *Phase ranged constraints and targeting of a nodesets*

```
<son name="IntCast">
    <targets>
        <target mandatory="no">
            <set name="Expression"/>
            <phases>
                <range from="phase1" to="phase3" />
            </phases>
        </target>
        <target mandatory="yes">
            <node name="IntConst" />
            <phases>
                <range from="phase4" to="phase7" />
            </phases>
        </target>
    </targets>
</son>
```

2.4 Strenghts And Weaknesses

....

Reordering child nodes causes create function to shift args

TRAVcont doesn't check for null

CHAPTER 3

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 abstractian 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 definiatly 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 plaform 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 **??**. 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.

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 **??**.

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]

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

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 Mechanisms

This section reviews common traversal and manipulation mechanisms applicable for trees with nodes in an inheritance tree and presents three traversal mechanisms for the type-safe AST. The section concludes with the benchmarking and optimization of the general purpose visitor and the lambda-based visitor.

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 seperating 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 upcomming three visitor implementations are a maximumum reduction of clutter and a transparant 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 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 **??**. 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 **??**).

3.3.2 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 propererty. 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 if always used for method selection. However, user code is bound to the single-dispatch mechanism which selects a method is based on the type of the *reference* holding the object. A derived type inside a variable of the base type will be routed to 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.

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 seperate 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 alltogether.

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 **??**.

3.3.4 Benchmarks & Optimization

The two traversal mechanisms compared in this section are the general purpose visitor and the lambda-visitor. A small benchmark suite has been developed which provides somewhat realistic data structures and can easily run tests on any number of visitor implementations. The view-only visitor was not included as it does not perform the same tasks as the other two.

First of all realistic test data must be produced. Part of the unit test suite included in the library of the framework provides static methods which generates small trees from a 10-node AST. Abstract classes are used as markers and the diversity of the nodes meets the criteria used in unit tests making them suitable for use in a benchmark suite.

The idea is to start with a small tree (n=14) with three layers of depth, and perform replacements with the same tree on certain blocks, providing a growth factor of roughly 5.5. After 4 iterations of the visitor the tree will have grown to 12098 nodes, which is likely below the limit where memory allocation starts to play a major role and above the threshold for measurable differences.

The first test run with 250 iterations, repeated 3 times produced the results shown in figure 3.1. Well, at least the performance impact of the `DynamicInvoke(...)` call are not quite as significant as some sources proclaim [7]. It does beg the question if the results should be explained by a largely similar algorithm, or if there is another bottleneck in the system.

The tests were run again and profiled using an educational copy of the JetBrains dotTrace profiling software. The number of iterations was brought down to 200 and the number of tries to 1. After 50 seconds the results were in: 42 seconds were spent inside in the system libraries on 700 million library calls. A mere 6 seconds were spent on all benchmarking code (responsible for 5.5 million node creations 30 million fixture calls), and less than a second was spend on garbage collection. The culprit was quickly identified: over 3 million cold calls were made to the extension method which enumerates the properties on a type which have the child attribute. These lists were cached, but on the objects themselves; perhaps a static dictionary is better suited. The results were drastic: the profiler ran in only 1.9 seconds, a reduction of factor 20. The first run was repeated and saw an factor 10 reduction in runtime (figure 3.2).

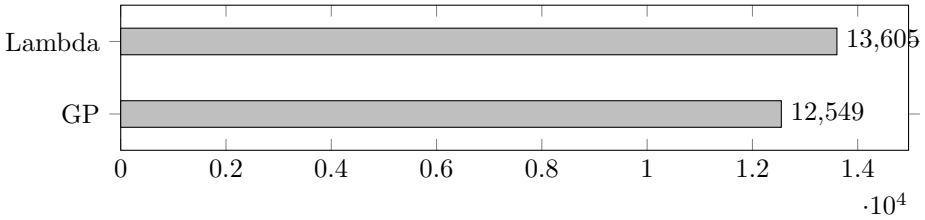

Figure 3.1: *Initial run* - avg. miliseconds per run, iters=*250,* n=*3*

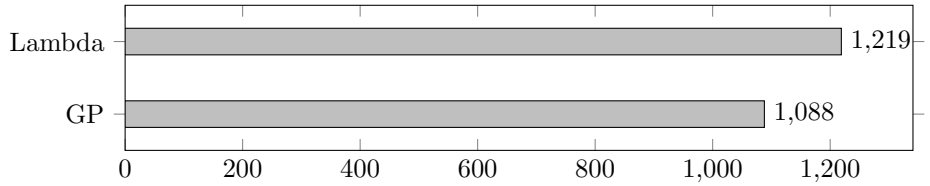

Figure 3.2: *Second run* - avg. miliseconds per run, iters=*250,* n=*3*

3.4 Pluggable Architecture

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

Phases reduced to list of traversals, as there is no need for logical seperation. - 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?

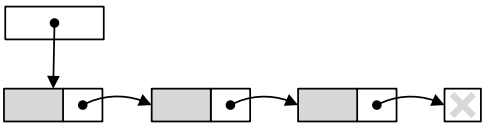

Figure 3.3: *Singly linked list*

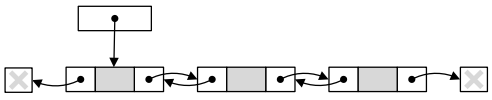

Figure 3.4: *Doubly linked list*

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. 3.3) or a doubly-linked list (fig. 3.4). A node holding any number of child nodes might be represented in C# as in snippet **??**.

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

Figure 3.5: *Dynamic array or list structure*

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

CHAPTER 4

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.

CHAPTER 5

Code Samples - To Remove

CHAPTER 6

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, 1993.