Computer Science — University of Amsterdam

BSc. Computer Science

University of Amsterdam

An object-oriented implementation of the CiviC compiler framework

Floris den Heijer
5873355

June 27, 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 8-week course where the main deliverable is a working compiler for a model programming language named CiviC, or *Civilized C*. This language exhibits characteristic features of structured imperative programming languages and is designed in a way to be sufficiently restricted in size and complexity for the course. Students may optionally implement more advanced features such as arrays and nested functions for extra credits. Construction is assisted by a framework which offers structured approach to compiler development.

The framework functions as a template, providing the necessary infrastructure and hooks for further development. It is written in C, a language most students are familiar with. It can be considered succesful, as nearly all students deliver working compilers with it and the course is received well. A recurrent problem during the first weeks of the course is the difficulty in getting up to speed with the framework. This is in part due to a lack of documentation, but it is suspected language plays a role as well.

A recent large-scale study found that features in a programming language considered most important were object inheritance, classes, exceptions and garbage collection[5]. Although the study might be biased towards users of dynamic programming languages, over 60% of respondants found static typing to be of at least medium importance. The framework's language (C) delivers little to none of these features. While it is statically typed, it is generally considered a 'weakly'-typed language compared to modern object-oriented languages. Languages such as Java and C# have more rigorous type checking and provide mechanisms to catch runtime violations of the type system. Static types have been demonstrated to serve as a source of documentation and may help programmers to use a new set of classes[3].

While C is unrivalled for it's high-speed, is it the right choice for the CiviC compiler framework? Does the framework utilize techniques which are only possible in C? Are there features made obsolete by simply using a modern statically-typed and object-oriented language? This paper will present an in-depth analysis of the CiviC compiler framework, documenting it's architecture, workflow and implementation. A new framework with the same goals as the existing one will be presented, but written in C#. Both approaches are rated on usability and maintainability, and the question *which is better* is discussed.

1.2 Context

- Discuss course and objectives
- Explain setting of framework

- Explain purpose of framework in more detail
- Compare against other frameworks

The CiviC compiler framework is used in a strictly educational setting and succesfully assists students in creating a fully functional compiler for the CiviC model language. and are there any comparable alternatives?

The course is structured to gradually walks students through all stages of compiler design. The first weeks are centered around frontend stages, such as AST design, scanning, parsing, semantic analysis and type checking. Only in the last few weeks are students introduced to the language they are compiling to, which is an assembly language for the CiviC-VM.

and very little attention is spent on heavy backend stages like code generation and optimization. This focus is not a bad thing, as the approach clearly works, delivering passing students with functional compilers every year.

As for other compiler frameworks, there aren't many general purpose versions about. The most well-known example might be the LLVM compiler infrastructure project[4], though it primarily fulfills a backend role by providing the specification for it's intermediate representation (IR) and a suite of backends targetting nearly every instruction set. Frontends targetting its IR are written by the community for a variety of languages, including Haskell and Python. Most other frameworks fill specific hardware niches such as GPGPU transpilers or optimizers for embedded IPC's.

There is one framework which stands out: Polyglot[8]. It has been primarily developed to assist in creating compilers for languages like Java, Java language extensions, implementations of of domain specific languages and for "...simplified versions of Java for pedagogical use". It's architecture and design choices will be considered in future chapters.

1.3 Research Questions

The main questsions of this project are:

> Does the framework offer features which a modern OO-language provides for free?
>
> How could a framework of similar purpose be implemented in C#?
>
> How do the approaches compare in terms of usability and maintainability?

Under 'framework of similar purpose', a framework is understood which:

1. Can be used to construct a CiviC compiler during the 8-week course
2. Provides a structured approach to compiler construction

1.4 Research Approach

To answer

> Does the framework offer features which a modern OO-language provides for free?

an in-depth of the CiviC framework will be given documenting it's architecture, workflow and technical underpinnings. Combined with knowledge of C#, the question can be objectively answered.

To answer

> How could a framework of similar purpose be implemented in C#?

the design process of such a framework will be documented. Specifically, the topic of type-safety and it's effect on key framework features will be discussed.

To answer

> How do the approaches compare in terms of usability and maintainability?

it is necessary to define usability and maintainability. While methods exist which quantify usability and maintainability to some degree, they are geared mostly towards rating enterprise software or based on large-scale surveys. The CiviC framework and it's userbase are too small for these methods, however it's size does warrant a more subjective approach.

Usability will be determined by comparing:

Clarity of user code Reference code will be judged primarily on conciseness. There are obviously major differences between C and C#, but it might be precisely these differences which makes one framework more accessible.

Transparency of framework Without detailled knowledge of the framework, how easy is it to understand how it how it functions and how user code fits in? Although it is not necessary to understand framework mechanisms in detail, users should have a rough idea of how it helps them; if not for debugging then for extensibility.

Maintainability is primarily relevant for those maintaining the framework, as most CiviC compilers will never be used outside of the course. A rough measure can by established by comparing the size of the code base.

1.5 Scope And Limitations

This paper is limited to the compiler framework of the CiviC toolchain and will not review the applicability of the the Civic VM and assembler. Teaching objectives or the suitability of alternative target languages such as LLVM, CIL or Java bytecode are also also outside of the scope for this project.

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 chapter concludes with a discussion on the

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.

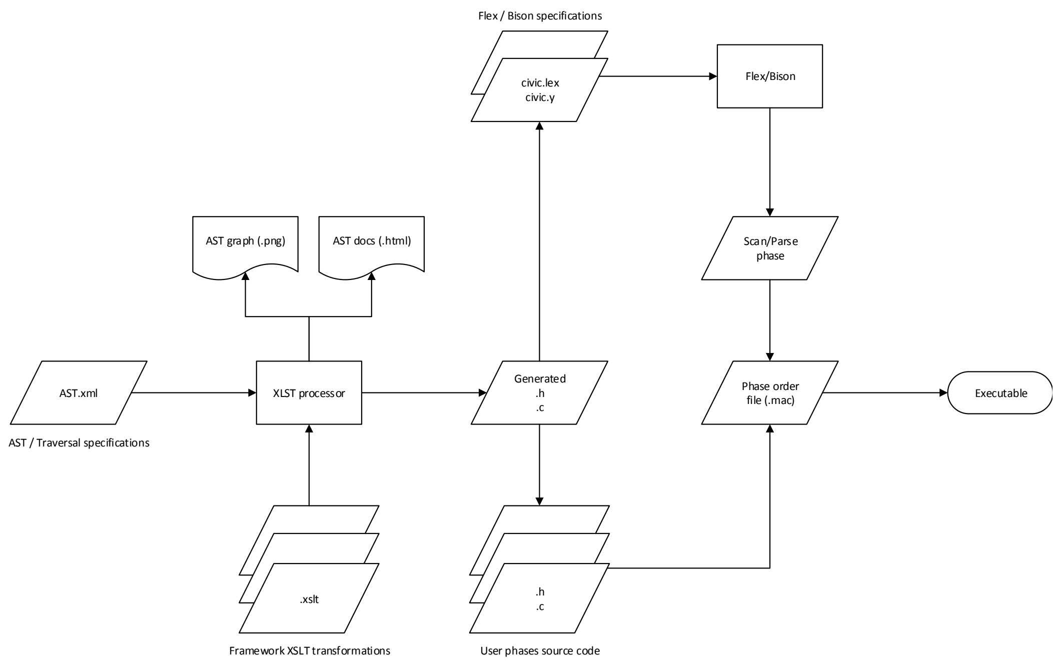

Figure 2.1: *Schematic overview of the Civic framework*

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 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. The command line also allows configuration of tracer options, which - assuming the user supplemented their code with the required statements - enables 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.*

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

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 under the `<syntaxtree>` inside 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. Passing parameters to nested traversals would require a seperate entry point with accompanying parameters. 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.

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

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

Attributes are copied and freed based on their specification in the previously mentioned `<asttypes>` 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.

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 Features Found In OO-Languages

The framework provides a wide range of features applicable for compiler construction. Some of these features seem at least partially focused around providing a level of type-safety and object-orientation not found in C.

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

Tagged union Nodes in the framework are represented by the `NODE` structure which, depending on node type, holds two other structures with child nodes and attributes. This can be seen as a primitive form of object inheritance, where the node structure functions as an abstract base class and all nodes as concrete implementations. The `NODE` struct is abstract because it holds a variable with an enumeration of all possible nodes, which means it's not possible to construct a node from code alone without also modifying that - generated - enum.

Access macros The access macros are obsolete if a node were to be represented as an object, instead of a composition of structures. Although the macros do not check for correct type due to their use on both sides on both sides of an assignment, an OO-language would provide this additional type-safety for free.

Memory management The most obvious feature most modern OO-languages offers is automatic memory management. The *free*-traversal can be removed entirely if a language with garbage collection is used. Similarly, object construction is done on a much higher level.

Validation traversal The *check*-traversal is primarily used to ensure nodes have valid children. If child nodes were to be typed properties of an object, this responsibility is handed to the compiler. The *nodesets* defined by the framework are only used in this context, and offer a limited form of inheritance. Inheritance doesn't allow nodes to belong to multiple nodesets directly, which is allowed by the framework; however the framework doesn't allow nesting of nodesets, which *is* allowed by inheritance.

Info structure Passed along with every traversal action is a traversal specific `INFO` structure, whose sole purpose is the sharing of state between traversal methods. If a traversal were to be seen as a class, this state could be transfered to member variables of the class.

CHAPTER 3

Re-Design In C#: Code-First Approach

Expanding on section 2.4, this chapter documents the design of an object-oriented framework centered around the same principles as the C-based framework. First the choice of C# is explained, along with a small primer on some of it's unique features. Then, the applicability of an AST abstraction layer as found in the current framework is challenged by developing a model for a type-safe AST. Development and optimization of two type-safe traversal mechanisms is covered next and the chapter concludes with a model for a code configured compiler pipeline and a discussion on representing collections in an AST.

3.1 C# Primer And Motivation

When considering imperative programming languages which are statically-typed, object-oriented and have automatic memory management the most obvious candidate would be Java. Most students have experience with Java as it is part of many other courses. So why C#? This section briefly discusses unique features of C# and why it is prefered over Java. It also doubles as a primer for readers not accustomed to it.

var keyword Writing staticaly-typed code can be a verbose affair. Initializing a class and assigning it to a variable in Java means typing the class name twice: once for the type of the variable, once for the calling the constructor of the class. This is tedious work and doesn't enhance readability by any stretch of the imagination. C# 3.0 introduced the `var` keyword, which infers the type based on the assignment to it. It can only be used in function scope and requires a direct assignment.

```
// Without var:
SomeExtremelyLongClassName onlyRelevantData = new SomeExtremelyLongClassName();
foreach (SomeExtremelyLongClassName item in ...)
    ...

// With var:
var onlyRelevantData = new SomeExtremelyLongClassName();
foreach (var item in ...)
    ...

var a = 5.0;        /* equivalent to float a = 5.0; */
var b = null;       /* illegal, type cannot be infered */
```

Auto-implemented properties Defining publicly variables on a class is 'bad-form' in the object-oriented world for many reasons. Exposing a member variable means the declaring class effecively loses control over it as external code can easily set it to `null` or a new reference. Control to variables is often wrapped with get/set methods, which return or set the member variable. This also allows code to mark data as read-only by only exposing a getter. In Java these methods must be written by hand, resulting in often tedious work or a task performed by the IDE. C# 3.0 introduced automatic properties, a compact and flexible way to define a private field and how define it's access logic. Properties can be initialized when constructing a class with an initialization block, eliminating the need for a constructor in most simple objects.

```
class Person
{
    public string Name { get; set; }
    public string NameInCaps
    {
        get { return Name.ToUpper(); }
    }

    public int Id { get; private set; }
}

var peter = new Person
{
    Name = "Not Peter"
};

peter.Name = "Peter";                 /* sets Name property */
Console.WriteLine(test.NameInCaps);   /* prints "PETER" */
peter.Id = 5;                         /* error, id can only be set from private scope */
```

Dynamic cast The `dynamic` keyword introduced in C# 4.0 is similar to the `var` keyword introduced above, except instead of defering resolution to compile time, the type is resolved *at runtime*. It allows constructs normally found in dynamic languages, such as duck typing, though this is of little use to this project. Objects can also be cast to the dynamic type, which extends C# further with dynamic dispatch. In the example below the C#'s single-dispatch and dynamic dispatch are demonstrated.

```
void PrintIt(Object o)
{
    Console.WriteLine("It's an object!");
}

void PrintIt(string s)
{
    Console.WriteLine("It's a string!");
}

object test = "Hello there";
PrintIt(test);              /* prints "It's an object!" */
PrintIt((dynamic) test);    /* prints "It's a string!" */
```

Anonymous types While Java allows anonymous classes but still requires the definition of a public interface these classes adhere to. C# 3.0 provides a mechanism for the construction of classes which encapsulate read-only properties without the definition of a type. Why this is so useful is demonstrated later, but for now it's just a convenient way to pack properties in a type-safe manner.

```
var p = new { Name = "Peter", Age = 25 };
Console.WriteLine(p.Name);       /* prints "Peter" */
Console.WriteLine(p.Age * 2);    /* prints 50 */
```

True generics Though C# and Java both have generics, there is a big difference in how they are implemented. Java generics use type erasure, which means a generic class is not actually generic, but operates on Objects uses casts to achieve type-safety. In effect they are a compile time construct and not available at runtime, which has significant implications on reflection and constraints. Generics in C# are supported at byte-code level by the *Common Language Runtime* or CLR, which allows deep type-checking and more detailled reflection[11].

Generic constraints in C# can specify a type argument to adhere to a specific inheritance relationship, such as a mandatory base class or other generic parameter. They can also specify a parameter to be a value-type or reference type[1], or that the reference type supports parameterless construction. Generics are supported on classes, interfaces, delegates and methods. Interfaces and delegates additionally support covariance and contravariance in their generic type arguments with respectively the `out` and `in` keywords. A short example is given below.

```
class Animal { }
class Fish : Animal { }
class Salmon : Fish { }

class Bear : Animal, IEatsAnimal<Fish> { }

interface IEatsAnimal<in T>
    where T : Animal
{
}

void FeedAnimal<TPred,TPrey>(TPred who, TPrey food)
    where TPrey : Animal
    where TPred : IEatsAnimal<TPrey>
{
}

var someAnimal = new Animal();
var salmon = new Salmon();
var bear = new Bear();

FeedAnimal(bear, salmon);      /* compiles, as salmon is convertible to fish */
FeedAnimal(bear, someAnimal);  /* fails to compile, someAnimal is not a fish */
```

Lambda expressions Lambda expressions are inline, local functions which can be passed as a function argument or returned as the value of a function call. They come in two variants: expression lambda's and statement lambda's. Both follow a similar construct, with a definition of input parameters enclosed with parentheses, followed by the `=>` lambda operator, followed by either an expression or a series of statements enclosed with braces.

They are different from *delegates*, in that a delegate represents a method's *type* by defining it's parameter list and return type as well as reference to the method. In a way they are a type-safe variant of function pointers in C. Lambda's can be used to *create* delegates as well as expression trees, though expression trees won't be covered in detail. A lambda can is convertible to an expression tree and can be compiled back to a delegate, allowing dynamic modification of executable code.

Lambda's can be assigned to any delegate, but not to an implicit local variable (i.e. `var`). The .NET framework defines a number of generic delegates specifically for use as anonymous

[1]The CLR categorizes types in these two classes; all primitives such as *int*'s, *float*'s and *struct*'s are value-types and everything derived from *Object* is a reference type, such as *class*'es but also *string*.

delegates. A function with a `void` return type and no arguments is represented by the `Action` delegate, a `void` function with one argument by the `Action<T>` delegate, etc. Functions with non-void return types are represented by the `Func<TResult>` delegate when they take no arguments, `Func<T, TResult>` if they have one argument, etc.

```
Func<int, int, int> add = (a, b) => a + b;
int result = add(2,5);      /* result = 7 */

Action<string> print = (s) => Console.WriteLine(s);
print("Hello");             /* prints "Hello" */
```

Extension methods Extension methods are in essence syntactic sugar for static methods, allowing them to be accessed 'as-if' they were instance methods. It is mainly a tool to increase readability of code when consuming classes from another project. Extension methods cannot access internal class variables and must be declared on a static class. The type they 'extend' must be their first parameter and must be preceeded by the `this` keyword, somewhat similar to how class functions are defined in Python.

```
static class Extensions
{
    public static void SayHello(this Ext.Object obj)
    {
        Console.WriteLine("External object says hello");
    }
}

var externalObject = new Ext.Object();
externalObject.SayHello();    /* prints "External object says hello" */
```

LINQ LINQ, or *Language Integrated Query* is a set of extension methods which operate on the .NET collection interfaces, providing native data querying capabilities to C# [10]. LINQ calls can be chained in a *fluent-style*, making them highly integratable. They use many of the techniques described above, building on extension methods, lambda expressions, generic delegates and anonymous types. It provides a default implementation on all CLR[2] objects, but allows data providers to implement part or the full range of methods. This has resulted in providers for SQL, XML, JSON, expressions, and many more.

An example of a LINQ interface is `IEnumerable<T>` which defines a class can be enumerated to return objects of type `T`. By using extension methods which operate on the enumerable interface, a class doesn't have to implement anything more than the basic functionality it provides. The most frequently used LINQ methods are documented in the next sample. All C#.NET collection classes like `List<T>` and `Dictionary<TKey, TValue>` implement the LINQ interfaces. They integrate very well with anonymous classes.

[2]Common Language Runtime

```
var ints = new List<int> { 1, 2, 3, 4, 5, -2, -3, 0, 0 };

// Projection:
//   .Select(Func<int, TResult> func) -> IEnumerable<TResult>
ints.Select(i => i*2);          /* returns enumerable of 2*i */

// List, array, dictionary conversion
//   .ToList(), .ToArray()
//   .ToDictionary(Func<int, TResult> key) -> IDictionary<TResult, TKey>
ints.ToArray();                 /* returns array of ints */

// Counting:
//   .Count(), .Count(Func<int, bool> predicate)
ints.Count(i => i % 2 == 0);    /* equals 4 */

// Tests:
//   .All(int => bool), .Any(int => bool)
ints.Any(i => i > 10);          /* returns false */

// And many more, some shown here:
var sum = ints
    .Distinct()          /* filter non-unique elements */
    .GroupBy(i => i%0)   /* group on even/uneven */
    .SelectMany(g => g)  /* flatten nested collection */
    .OrderBy(i => i)     /* ordering */
    .Where(i => i > 10)  /* filtering */
    .Sum(i => i*2);      /* summation */
```

3.2 Type-Safe AST

Working with a strongly typed OO-language grants the ability to model AST nodes in greater detail than before possible. The most intuitive way is to represent different types of node with distinct classes. All nodes derive at some point from an abstract base class, though intermediate classes can also be created to allow for hierarchical grouping. Functions can be specified on a subset of nodes by targeting a type anywhere in the inheritance chain. The attributes and child nodes should be exposed as typed properties. This setup guarantees type-safe access to nodes, and prevents users from placing an incompatible node as a child node. However, if we assume a traversal function to operate on a base class and declare child nodes as properties on more derived types, how can a traversal ever access them?

Listing 3.1: *Marking traversible child properties with an attribute*

```
class ConcreteNode : Node
{
    [Child] public OtherNode Child1 { get; set; }
    [Child] public OtherNode Child2 { get; set; }

    public string Attribute { get; set; }
}
```

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 and use reflection to retrieve them at runtime, as shown in listing 3.1. This method is impossible to implement in C, as it has no runtime let alone reflection.

Simply marking properties is not enough, as the order in which properties are declared on a class is not guaranteed to match the order in which they are returned from reflection. Luckily information such as declared line number can be accessed by passing a special parameter to the constructor of the attribute. This can be done oblivious to the user of the attribute. A function which iterates over the child nodes needs to order it's output based on this line number. This function also has another important responsibility, as attributes in C# cannot be constrained on type like generics can. This means a reader should check if the type of the property is *at least* convertible to the node base type, although it can be any derived type too.

The `Node` abstract base class and the `[Child]` attribute form the basis of the approach chosen for the type-safe AST. Nodesets from the previous framework can be exchanged for base classes to allow easy targeting of expressions, statements, or variable declarations. A small difference is that C# (and Java for that matter) only support single-inheritance, so a node can only ever belong to one inheritance branch. To support this scenario an implementation for a tagged union would have to be made, similar to the old framework. This complicates matters immensly, as for every function acting on a node or attribute all options have to be accounted for, and a system like pattern matching in functional languages would have to be implemeneted. As it is hard to find a legitimate reason to have a node belonging to two *unrelated* sets, this has not been pursued.

It must be noted that the order in which child properties are returned is ambiguous if 1) properties are defined on multiple classes in an inheritance chain and 2) these classes reside in different files. Properties are correctly retrieved but are sorted on line number, which is valid only for a single file. This behavior is undefined, but can be accounted for if a routine is written which orders by relative depth in the inheritance tree before sorting on line number. Not only is this non-trivial and costly to run, there are no valid cases for such an AST with the CiviC language. Even when accounting for inheritance order, C#'s concept of partial classes also break this mechanism. In other words, perfect ordering is possible when taken slight care when declaring nodes.

Equally difficult to implement was the idea that built-in *collections* could be used instead of linked-list structures inside the tree. Often the SLL container node would only be used for the purpose of maintaining that list, why not represent it as a generic list? It would certainly make tasks like appending statements or inserting variables a lot easier. To cut to the chase: experiments on such an approach resulted in code far too complicated for what it actually achieved, and the required functionality could be provided by a generic SLL template and an extension method.

The prototype of the `LinkedNode<T>` class and the extension method are listed in listing 3.2.

Listing 3.2: *Prototype of the LinkedNode<T> SLL container and extension methods*

```
public class LinkedNode<T> : Node
    where T : Node
{
    [Child] public T Node { get; set; }
    [Child] public LinkedNode<T> Next { get; set; }

    // Retrieves the last linked list node in the chain.
    public LinkedNode<T> Last();
    // Inserts node directly adjecent to this one.
    public void InsertAfter(T node);
    // Inserts a collection of nodes directly adjecent to this node.
    public void InsertAfter(IEnumerable<T> nodes);
    // Appends a node at the back of the linked list.
    public void Append(T node);
    // Appends a collection of nodes at the back of the linked list.
    public void Append(IEnumerable<T> nodes);
    // Enumerates all stored nodes in the list, excluding the containers.
    public IEnumerable<T> NodesInLinkedList();
}

static class Extensions
{
    // Convers a collection of nodes of type T to a chain of linked list nodes.
    public static LinkedNode<T> ToLinkedNode<T>(this IEnumerable<T> collection);
}
```

3.3 Improved Traversal Mechanisms

This section discusses ways in which a type-safe AST can be manipulated and presents two traversal methods exposed by the C# framework. The section concludes with a short optimization round of the two traversal methods.

3.3.1 On Visitor Patterns

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 base ‘Visitor’ class which contains a visit method for every node in the hierarchy. The visit method has a void return type and accepts an argument holding the concrete node type. A method accepting the visitor class is added to every node, whose sole purpose is to return control to the visitor class by calling the visit method for that node. This inversion of control works around compile-time method resolution, or single-dispatch, and creating a form of double-dispatch. Each new operation on the tree results in a new visitor based on the abstract visitor class, implementing the visit methods for every node type. The classic example of the pattern is not shown here, as it only demonstrates unwanted code.

In the classic visitor, each concrete visitor must implement methods for all node types, just like the user-mode traversals in the C framework. The first workaround for this problem is the ‘default-visitor’ drawn-up in 1996, which adds another layer of inheritance to the visitor inheritance tree[7]. The same paper explains another variant on the visitor: the ‘extrinsic visitor’, which relies on function lookup tables in combination with inheritance to call the correct visitor function, which is surprisingly close to the approach from the C-framework. A generalized pattern for all visitor patterns was published in 1998. The pattern, known as *Walkabout*, is explained in a mix of Java and pseudo statements. Understanding all visitor patterns is relatively easy by observing their template, so it has been included in listing 3.3[9].

Listing 3.3: *Walkabout program 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  }
```

The template describes a visitor base class called ‘Walkabout’, with a visit method acting on all node types (line 3), the most general form of which is - ofcourse - Java’s `Object`. If the node passed in isn’t null, a non-existent but defining method is invoked (line 6). How exactly it is determined that a public visit method has been defined for v is left unspecified, as it depends on the type of visitor. If there is a method defined, that method is invoked (lines 6-7), if not, all fields of the node are instead visited (line 8-10).

While the template deals with the basics of the pattern, it leaves much to be desired. It does not deal with or mention the possibility of more than one matching visit method for a node, something which is quite common and necessary behavior if inheritance is used in node composition. Another shortcoming is that it doesn't support return values, preventing such visitors from completely redefining a tree (i.e. a new root node is never possible). The C-framework did provide this mechanism, how can it be implemented in a type-safe language?

There seem to be no literature on this topic, which is an either an indication of unexplored potential or a sign that the concept is inherently flawed. We'll assume the first. A hint that visitor methods which return something is a viable concept is provided by the visitors generated by ANTLR v4, one of the most widely used scanner/parser generators[1]. In addition to supplying users of the with a default visitor, it exposes the `AbstractParseTreeVisitor<T>` which accumulates a return type. It does state the intent of this parameter to be of a different nature than the nodes in the tree, however the parse trees of ANTLR v4 are not meant to be manipulated like an AST.

As we're treading into unknown waters of visitor patterns, design goals for the visitor must be set. First of all, it should be as 'lean' and isolated as possible: an 'accept'-method on all nodes for every type of visitor is not desirable. It mixes concern in an attempt to work around a language feature, perhaps 18 years of language development have made this possible. Secondly it should be easy to understand and test.

3.3.2 Derivation Of The Generalized Visitor Pattern

Could the Walkabout program be modified to include return types and still result in a type-safe visitor? Basic requirements for such a visitor are:

1. The visitor must be an abstract base class with a publicly exposed method which starts traversal
2. Concrete visitors implement public visit methods for all nodes they want to handle
3. Concrete visitors may define visit methods for any node in the inheritance hierarchy

Explicitly left out is the definition of a method signature for visit methods. While it is clear they must always return something which comes in, how should a handler be defined which replaces a node with a less derived type, or a sibling?

Listing 3.4: *The main problem with return types and visitors*

```
1  class Node { }
2  class Derived : Node { }
3  class A : Derived { }
4  class B : Derived { }
5
6  class SomeVisitor : GeneralizedVisitor
7  {
8      public B Visit(A node)
9      {
10         return new B();
11     }
12 }
13
14 var visitor = new SomeVisitor();
15 A a = new A();
16 a = visitor.Visit(a);    /* what happens? what should happen? */
```

A troublesome situation is shown in listing 3.4. What actually happens is that the code *won't compile*. This is because the visitor invokation on line 16 is routed with single-dispatch to the concrete visit method (lines 8-11), and that method has return type B. Ideally, a situation is desired where this incompatible visitor method is not invoked and the method on the base class is invoked, performing some sort of appropriate method lookup. This method has been left unspecified, but if were to be defined as a method taking a node of type `<T>` and returning something of the same `<T>`, the situation would still be the same.

Matching methods on return type is a form of multi-dispatch which is unavailable in C#. It can however be at least partially simulated with reflection. Unavoidable is the impossibility of defining two visit methods with the same argument, differing only on return type.

More importantly, a restriction on types must be defined for transformations. If we wish to transform a node of type `X` to a node of type `Y`, with both nodes sharing a base type S; the return type of the visitor method should be `S`. This type `S` is at the very least of type Node in the case of no intermediate base classes. When multiple base classes match, the best choice is often the most derived base.

This would change line 8 of listing 3.4 to:

```
public Derived Visit(A node)
```

This doesn't change the fact that the sample won't compile. For it to do so, we must invoke the - still unspecified - base method. This can be done by using the same weakness inherent to single-dispatch the visitor pattern was designed to circumvent, which is instead of defining `a` to be a variable of type `A`, instead define it as a `Node` or alternatively `Derived`. This results in a change of line 15 to:

```
Derived a = new A();
```

Basically ensure that the call is routed to the base visitor method, regardless of the reference type. This 'hack' isn't as bad as it seems as will be shown later. Back to the base method, it *cannot* be defined in terms of `Node`'s, because that would imply a type `Node` would be returned. If such a method were to be defined, the result of the method call would need to be cast back to `Derived`, which is insanity as this the node passed in is already more general than the actual type. When invoking the base method we only want a node back of the exact type passed in. This results in a base visit method defined as:

```
T Visit<T>(T node) where T : Node { /* return some T */ }
```

At least the code compiles, finally the Walkabout program can be implemented in the base class! Unfortunately, this is not the case. First of all, while the base visit method may be invoked with a node of type `T`, or more acurately, *of compile-time* `T`, we've just explicitly stated that in order to avoid some situations it is acceptible to cast it to something more general. This means the base visitor method shouldn't just return 'some type T', but instead should return 'some type R where R is the runtime type of the input'.

This definition simulates the `dynamic` keyword explained in section 3.1. So why aren't we designing the visitor entirely around this magic runtime resolved cast keyword? Well, there is one last piece of the puzzle not in place: automatic traversals. In the Walkabout program from listing 3.3 this is handled in lines 9 and 10. What this code doesn't address is the fact that the return value of the visit call should be assigned back to the property. In pseudocode:

```
for (each field f of v)
    v.f = this.visit(v.f);
```

This modification is crucial, because the compiler isn't smart enough to forsee incompatible types and instead of a compile-time error will throw a runtime exception. To warrant a mechanism other than a dynamic cast, consider that there is no way to *enforce* the format of a visit method defined earlier, that is, a visit method should return something which shares a base class with the node passed in. To work with the example from listing 3.4, this issue can be demonstrated by introducing another node `C` which is derived directly from `Node`. Now consider a property f of type `Derived` on some v, and a publicly accessible visit method which accepts type `C` and returns type `Node`. Dynamically casting the value of field f results in a match on the visit method, which returns some `Node`. Assigning this value to f throws a runtime exception.

The type of the result from the visitor could ofcourse be checked, but only *after* the method has been executed! The visitor method and expected the ability to return a `Node`, adjusted the state of the visitor accordingly, only to have it's results discarded. This situation should be avoided at all costs, since bugs of this type are near impossible to trace.

What is instead proposed is the use of reflection to resolve the method, but make sure the return type is convertible to the property type before execution. This constraint on the return type should be passed as an additional argument to the base visit function. It should however be hidden from concrete visitors, as they already have a workaround which aborts compilation if improperly implemented. This can be achieved by making the base visitor method with the type constraint parameter private, and wrapping it in a public base visitor method without the additional parameter. For added generality, the base class has been made generic on the root node. Translated back to pseudo-C#-Java, the template for the generalized visitor is shown in listing 3.5.

Listing 3.5: *Generalized visitor pattern, in similar style to Walkabout*

```
class Generalized<Root>
{
    T visit(T v) where T : Root
    {
        return Visit(v, typeof(T));
    }

    private T visit(T v, Type max) where T : Root
    {
        if ( v == null )
            return null;

        if  (/* this visitor has a public visit method for the runtime type of v */)
        and (/* the return type of this method is equal to or a subclass of max  */)
            return (T) invoke(method(v));
        else
            for (each field f of v, with type ft)
                v.f = Visit(v.f, ft);
    }
}
```

3.3.3 The Generalized Visitor

Using the template from listing 3.5 and the type-safe AST from section 3.2, the generalized visitor can be easily constructed. Four pieces of code should be assembled:

1. Code which finds a public visit method applicable for a node n of type T with runtime type R
2. Code which ensures the return type of this method is equal to or a subclass of type M
3. Code which returns and casts the invocation of the method, if any
4. Code which handles traversal of children

The first piece is easily implemented using the `Type.GetMethod(string, Type[])` reflection call, invoked on the type of the visitor. The current runtime type for a class is obtained by calling `Object.GetType()`, or simply `GetType()`. The name of the method we're interested in is "Visit", which has a single parameter: the runtime type of the input node.

```
MethodInfo method = GetType().GetMethod("Visit", new[] { R });
```

Moving on, the heavy lifting which ensures the method has a convertible return type is done by using the `Type.IsSubclassOf(Type)` method, resulting in the following check:

```
if (method != null && (method.ReturnType == M || method.ReturnType.IsSubclassOf(M)))
```

Invocation and casting of said method is equally painless. The `MethodInfo.Invoke(object, object[])` call invokes the method refered to by the `MethodInfo` instance on the object passed as the first argument. An array of objects passed as the second argument to the function, in this case an array of the input node n.

```
return (T) method.Invoke(this, new object[] {n});
```

The code responsible for traversing the children of a node is perhaps better fit for a new function `VisitChildren(Node)`, which can be made publicly available. It is after all desirable to be able to continue traversal of child nodes once inside a visit function. The code responsible for generating a list of `PropertyInfo` objects based on the `[Child]` attribute has been left out for brevity. The `VisitChildren(Node)` method has been listed in full as it is reused in the next section.

Listing 3.6: *Method responsible for invoking the generic visit method on child properties*

```
public void VisitChildren(Node node)
{
    if (node == null)
        return;

    // Traverse child nodes using attributes and replace with result from visitor.
    foreach (var prop in node.ChildProperties)
    {
        var value = (Node) prop.GetValue(node);
        if (value != null)
        {
            value = Visit(value, prop.PropertyType);
            prop.SetValue(node, value);
        }
    }
}
```

This concludes the implementation fo the generalized visitor. There are two limitations on this visitor which may or may not apply to users of the framework, as it depends on the design of the AST and implementation of a traversal. These are:

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

Why would anyone need two handlers with their only difference being the return type? Ignoring that for now, an abstract use case might be a handler which transforms according the transformation A -> A if the transformation A -> B does not 'fit' or apply. While the first case results in illegal C# code, the second case 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.

3.3.4 Lambda Visitor

While the general purpose visitor is sufficient for most traversals, it requires some care when targeting multiple layers of an inheritance hierarchy. Method invocation depends on two seperate mechanisms: single-dispatch and dynamic-dispatch via reflection. 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

There might be cases where the user simply wants to 'view' nodes of a certain type, without having to write handlers in block form with a return statement. As exhibited in section 3.1, .NET defines 'Action' delegates for `void` returning functions, and 'Func<TResult>' for functions returning `TResult`. With the ordered list approach these two types can be interleaved, as a `void` handlers can be assumed to return itself.

If the logic determining the handler is 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 concise traversal definitions. This adds the following to the requirements of the handler structure:

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.

Code inheriting utilizing the visitor must have some way of 'configuring' it, i.e. defining all handlers for the traversal. These configuration methods directly manipulate the list of handlers, and also provide type requirements on the lambda handler added to the list. Consider the basic case where a handler for a node of type T, that is, *runtime type* T is added. The handler will be defined as `void` returning delegate and can be configured by calling a configuration method:

```
void View<T>(Action<T> action)
    where T : Node;
```

Similarly, if the replacement of a node of type T with a node of type P is desired, with P and T sharing a common base TContext, the following configuration method is invoked:

```
void Replace<T, TContext>(Func<T, TContext> function)
    where T : TContext
    where TContext : Node;
```

Specifying this *context* also works on `View<,>(..)` methods. Adding a predicate is done by calling the `ReplaceIf<>(f, pred)` and `ViewIf<>(f, pred)` configuration functions. Also, as mentioned in section 3.1, lambda's are assignable to delegates; the reverse also holds true (to some extend): C# allows the binding of a method group to a lambda expression. An example demonstrating the different configuration methods provided in listing 3.7.

Listing 3.7: *Configuration methods of the lambda visitor*

```
View<IntConst>(i => Console.WriteLine(i.Value));    /* prints value of int const nodes */

ReplaceIf<IntConst>(i => null, i => i.Value > 10);  /* removes all int nodes > 10 */

Replace<Expression, Node>(ex => new Node());        /* intermediate type target, only
                                                       executed on expressions in a
                                                       property of at least Node */

/* Configuration of lambda with statements */
View<Node>(n =>
{
    Console.WriteLine(n.GetType());
    n.DoSomething();
});

/* Method which converts IntConsts to BinOp's through Expression... */
Expression TransformInt(IntConst i)
{
    return new BinOp(..);
}
/* ... can be added without specifying types, as they are infered from the method */
Replace(TransformInt);
```

The private base visit method for the lambda visitor is given in listing 3.8. The `VisitChildren(Node)` methods is the same as in the generalized visitor pattern.

Listing 3.8: *Implementation of the lambda visitor with dynamic invoke*

```
private T Visit<T>(T node, Type maxUpcast)
    where T : Node
{
    if (node == null)
        return null;

    // If a handler exist for this class with convertible return type, invoke.
    var nodeType = node.GetType();
    foreach (var handle in _handlers)
    {
        // Test for matching type (or subtype).
        if (!handle.Type.IsAssignableFrom(nodeType))
            continue;

        // Test for fitness of transformation result.
        if (!maxUpcast.IsAssignableFrom(handle.MaxUpcast))
            continue;

        // Test for predicate.
        if (handle.Predicate != null)
        {
            bool result = (bool) handle.Predicate.DynamicInvoke(node);
            if (!result)
                continue;
        }

        // Check if this is an action or function.
        if (handle.Function != null)
        {
            return (T) handle.Function.DynamicInvoke(node);
        }
        else
        {
            handle.Action.DynamicInvoke(node);
            return node;
        }
    }

    // If no handlers are found or if all fail, traverse children.
    VisitChildren(node);

    return node;
}
```

3.3.5 Benchmarks And Optimization

Completely eliminating the dispatch mechanism of C# is bound to have an effect on execution speed. However, both the generalized visitor and the lambda visitor use some form of reflection. This section compares performance of the visitors and will add several optimizations.

A small benchmark suite has been developed which provides realistic data and can easily run tests on both visitors. To generate somewhat realistic test data, fixtures were used from the projects unit tests which generate small trees build from a 10-node AST. The idea is to define a traversal which takes a tree and, through repeated invocation, causes an exponential growth.

A traversal was constructed suitable for both visitors with a growth factor of rougly 5.5. This means a small tree of 14 nodes with a maximum depth of 3 put through 4 vistitor iterations will have grown to over 12,000 nodes; an arbitrarily chosen number likely below the threshold where memory allocation starts to play a major role, and likely big enough to demonstrate a difference in performance.

The first test run with 250 iterations, repeated 3 times produced the results shown in figure 3.1.

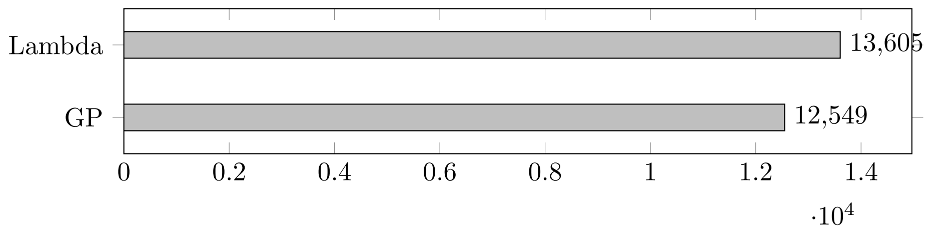

Figure 3.1: *Initial run* - avg. miliseconds per run, iters=*250*, n=*3*

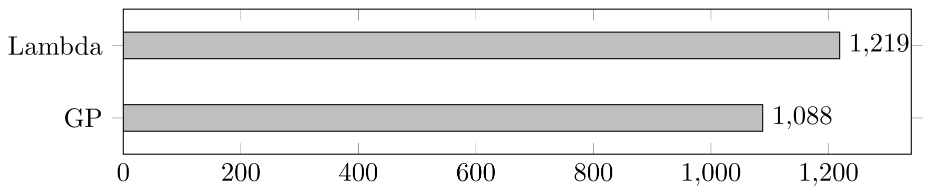

Figure 3.2: *Second run* - avg. miliseconds per run, iters=*250*, n=*3*

Well, at least the performance impact of the `DynamicInvoke(...)` call are not quite as significant as some sources proclaim [10]. 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 factor 20 reduction. The first run was repeated and saw an factor 10 reduction in runtime (figure 3.2).

3.4 Code-Configured Architecture

One of the features which translates well to the OO-framework is the phase based pipeline. To summarize section 2.3.3: the C-framework defined the compiler pipeline as a sequence of phases, each phase composed of one or more entry points. Running `make` flattened the sequence, piping the result of one transformation into the next until either compilation succeeded or stopped prematurely.

This structured approach is an essential part of compiler design and can easily be implemented in C#. The definition of 'entry-point' is formalized by the interface `IPhase`, which has only one method responsible for invoking underlying traversals. Phases must implement this interface if they are to be used in by the `PhaseRunner` control logic. Phases may optionally define an attribute class containing the name of the phase, which can be printed if requested.

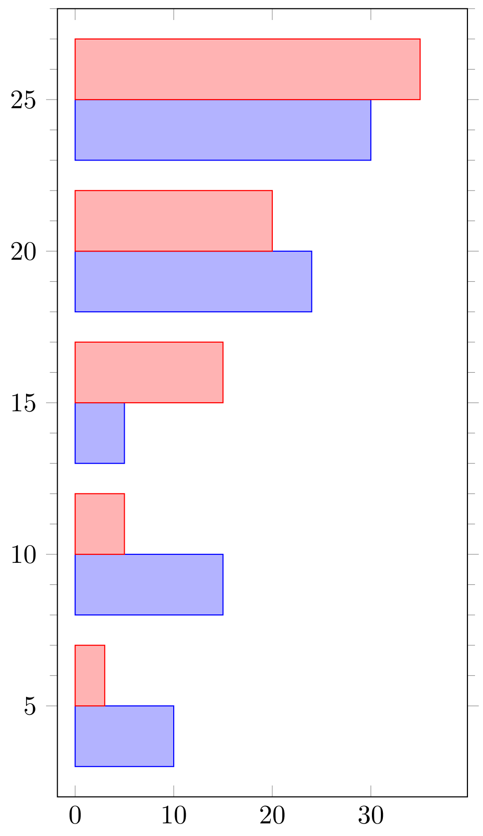

Figure 3.3: *Removing intial bottleneck*

Listing 3.9: *Phase entry point specification*

```
public interface IPhase
{
    Node Transform(Node node);
}

[Phase("Type Checking: Performs rigorous checks on types")]
class TypeChecking : Visitor, IPhase
{
    // Entry point
    public Node Transform(Node node)
    {
        return Visit(node);
    }

    // Visitor methods
    public Expression Visit(...)
}
```

Configuration of phases is done entirely from code, which allows configuration of phases based on command line arguments or (un-)commenting of certain phases during development. The grouping of entry-points in the C-framework has been removed in lieu of a single list of `IPhase` instances, demonstrated in listing 3.10. Execution can be aborted at any point during compilation, or halted if errors are found after a phase.

Listing 3.10: *Phase runner configuration*

```
// Create runner and register stages
var compiler = new PhaseRunner();

compiler.Phases.AddRange(new IPhase[]
{
    new ParsePhase(options.InputPath),
    new PrintTree(),
    new SymbolResolution(),
    new RenameUserSymbols(),
    new TypeChecking(),
    new ExpandArrayDimensions(),
    ...
});

// Run phases.
compiler.Run();
```

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 through built-in lists?

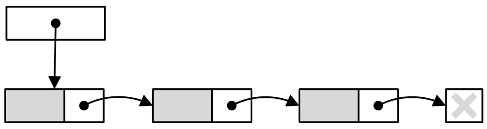
Figure 3.4: *Singly linked list*

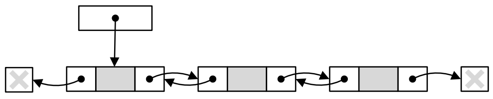
Figure 3.5: *Doubly linked list*

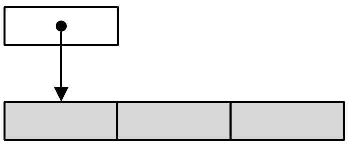
Figure 3.6: *Dynamic array or list structure*

A variable number of nodes is implemented in an AST by nesting container nodes up to the depth required, forming either a singly or doubly linked list (resp. figure 3.4 and figure 3.5). A node holding any number of child nodes might be represented in C# as in snippet listing 3.11.

Listing 3.11: *Example of lists as traversible attributes*

```
class FunctionDefinition : Node
{
    [Child] public List<VarDec> Variables { get; set; }
    [Child] public List<Statement> Statement { get; set; }
    [Child] public Expression Return { get; set; }
}
```

This poses the question how such a list property should be handled.

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.

CHAPTER 4

Usability And Maintainance Comparison

Ranking two different software packages is inherently subjective, though such an approach is actually warranted in this situation. The C# framework has been field tested by one group from the 2015 course, who managed to deliver a compiler with all optional features earning them one of the highest grades in the class. This proves at the very least that the C# framework can be used for the course.

4.1 Usability

While the lack of documentation might appear as a major blow to usability, a more pressing issue is found in the organization of the framework. The framework languages are none of those. While the transformations are well documented, the chance of anyone reading them are slim as they are tucked away in the framework directory, which is excluded from any user interaction. An example of this is the relocation of the scan/parse phase implementation file to the framework folder, even though a dedicated folder is available. This adds to the black box that is the build process.

Running `make` can be a scary thing. The roughly 300 line `Makefile` is grouped by function and spread out over five files and while certainly clarifying where you should add your code, it doesn't make it any more clear how the process actualy works. This is a shame, because the foundations are actually incredibly well laid out. So, it's not the lack of documentation per se, the process is intentionally obscured. One could say the code generation layer is also to blame. It takes 32 XSLT transformations to generate the code necessary for the most basic compiler.

Clarity of user code: conciseness

Transparency of framework (how easy it is to comprehend action)

So, where to start?

Maintainability (size)

Other features (discussion):

- NuGet support for package updates
- ANTLR4? (LL(*) with EBNF). Better libraries might be a point
- More relevant examples found with standard lib

4.2 Maintainability

Software maintainability is at least in part affected by size of the code base. Rough figures on code size were collected using the free tool LocMetrics, which computes various source lines of code (SLOC) metrics. Using lines of code for anything other than exploratory research is dangerous, as there is no scientific consensus on what universally qualifies as a line. However, there is some agreement on what is applicable for C-based languages.

When discussing lines of code, a distinction is generally made between the number lines in a file and the logical executable lines of source code (SLOC-L), with a logical line possibly spanning multiple lines. For C-based languages the logically executable lines of code metric is calculated with relative easy by simply summing all terminal semicolons and terminal curly braces[6]. However, logical lines of code *excludes* preprocessor directives and considering a sizeable part of C-based framework is in some dependant on macros, a broader definition is necessary. LogMetrics defines the physical executable lines of code (SLOC-P) as all lines in a file, minus blank lines and comment lines. These three metrics combined should give a clear indication on the size of the projects.

One problem in assessing the size of the C-based framework is in the XSLT transformations and generated code. XSLT is a programming language in itself, is Turing complete and should at least in some way be integrated. No SLOC measures are known for XSLT, instead a figure for the raw lines was chosen. The transformation sheets contain some documentation, so 25% has been deducted from the raw number of lines. Rather than measuring the sum of generated code and the framwork base code, two measures were used: one for the base code without generated code (no-gen), and one for purely the generated code (gen-only).

The C# framework has a seperate project with unit tests, these have not been included with the reference metrics but are recorded seperately.

Framework		**Lines**	**SLOC-P**	**SLOC-L**	**Files**
C (*.c; *.h)	*no-gen*	9800	3800	2200	40
C (*.c; *.h)	*gen-only*	8500	5300	2900	21
C (*.xsl)		5000	N/A	N/A	32
C# (*.cs)		1350	980	560	19
C# (*.cs)	*unit tests*	830	620	355	10

Table 4.1: *Framework size metrics*

CHAPTER 5

Discussion

Nope.

CHAPTER 6

References

[1] Antlr v4, 2015. URL http://www.antlr.org/.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[3] 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.

[4] Chris Arthur Lattner. *LLVM: An infrastructure for multi-stage optimization.* PhD thesis, University of Illinois at Urbana-Champaign, 2002.

[5] 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.

[6] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. A sloc counting standard. In *COCOMO II Forum*, volume 2007, 2007.

[7] Martin E Nordberg III. Variations on the visitor pattern. *Ann Arbor*, 1001:48108, 1996.

[8] Nathaniel Nystrom, MichaelR. Clarkson, and AndrewC. Myers. Polyglot: An extensible compiler framework for java. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-00904-7. doi: 10.1007/3-540-36579-6_11. URL http://dx.doi.org/10.1007/3-540-36579-6_11.

[9] 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.

[10] Joel Pobar. Dodge common performance pitfalls to craft speedy applications, 2005. URL https://msdn.microsoft.com/en-us/magazine/cc163759.aspx.

[11] Jonathan Pryor. Comparing java and c# generics, 2007. URL http://www.jprl.com/Blog/archive/development/2007/Aug-31.html.