ptr-tidy: Automatic refactoring of raw pointers in C++

Artem Usov
2296905U

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Masters project proposal

18 December 2020

Contents

1 Introduction

1.1 Difficulties with Memory Management

Systems programming often involves memory management, that is requesting memory from operating system to be used and managed by the program. This is done when the amount of memory that we need cannot be known at the time of compilation of the program. However doing so creates the opportunity for memory safety errors [5] which are notoriously challenging to avoid.

How challenging one might ask?

In a presentation by Matt Miller, a security engineer at Microsoft, it is shown that around 70% of their vulnerabilities that are addressed through a security updates are due to memory safety issues [18]. In another presentation at the Linux Security Summit it is shown that in several other top projects such as Firefox, macOS, Ubuntu and Android all had over half of their CVEs [1] attributed to issues with memory safety [10]. This is at industry-leading companies who are renowned for hiring top talent. Such memory safety vulnerabilities can be exploited, and due to new regulations such as the GDPR, these issues are more commonly exposed to the general public and punished by regulators. An example is a fine of £20m for a British Airways data breach by the British Information Commissioner's Office [19].

1.2 Possible Solutions to Memory Management

Memory safety errors occur in languages that place the task of memory management with the programmer, such as C and C++. However simply not using these languages is not an option since their unmanaged nature makes them highly performant, and therefore the best option for systems such as a web browser.

What can be done then to address this issue?

Engineers at both Microsoft [25] and Mozilla [11] converge on Rust [1] as a possible solution. Rust is a systems language that offers similar performance as C and C++ [16], however its linear type system and memory ownership model also guarantee memory safety. The rewriting of a program in a new language, namely from C++ to Rust, is a colossal undertaking, especially given that Rust has a reputation for being difficult to learn. We therefore propose an alternative partial solution to this problem.

[1] `https://cve.mitre.org/`

```
1 void foo(int x) {
2     Shape *p = new Circle{Point{0,0},10};
3     // ...
4     if (x<0) throw Bad_x{}; // potential leak
5     if (x==0) return; // potential leak
6     // ...
7     delete p;
8 }
```

Listing 2.1: Example of memory leaks using manual management. Lines 4 and 5 show how the memory allocated on line 2 will never be freed since the control flow of the function is interrupted before we reach line 7 and free the memory

2 Statement of Problem

2.1 Using Smart Pointers for Memory Management

Generally pointers are used to give the program access to a resource that cannot be directly included in the program itself, such as allocated memory or a file. However Stroustrup claims that pointers to objects allocated on the free store are dangerous and a *plain old pointer*, or raw pointer as we will refer to them, should not be used to represent ownership [23].

In Listing 2.1 we can see that when using a raw pointer, there are two cases in which the programmer will never free the allocated memory. Instead we can use *smart pointers* [6] and the concept of *RAII (Resource Acquisition Is Initialisation)* [22] to create resource handles which automatically eliminate resource leaks with no, or very little added overhead. The first type of smart pointer are *unique pointers* which have only a single unique owner at any time. We can therefore fix the previous example in which we leak memory by using a unique pointer, as can be seen in Listing 2.2. In the comments of Listing 2.2, we can see all the locations where the control flow exits the scope where p is defined and where the compiler will automatically insert delete statements, now ensuring that the resource will never get leaked. The ownership of a unique pointer can be transferred between variables using `std::move` which moves the ownership from one variable to another, making the original reference invalid.

The second type of smart pointer are *shared pointers* which are used for resources which may not necessarily have a single unique owner and so several owners *share* ownership to the resource. Reference counting [4] is used to ensure that after the last owner loses access to the resource, it will finally be freed. Shared pointers are suitable to be used in most situations and such reference counted pointers are used by default for all values in the Python and Swift languages.

```
1 void foo(int x) {
2     std::unique_ptr<Circle> p = std::make_unique<Circle>(Point{0,0},10);
3     // ...
4     if (x<0) throw Bad_x{}; // delete p inserted here
5     if (x==0) return; // delete p inserted here
6     // ...
7     // delete p inserted here
8 }
```

Listing 2.2: Example of using a unique pointer to manage memory. Note how the programmer no longer has to specify to manually delete the pointer as in Listing 2.1, and how the compiler can detect that on lines 4, 5 and 7 the variable p leaves the scope and therefore needs to be freed. The compiler will automatically insert these delete statements during compilation.

2.2 Do Smart Pointer Solve Memory Management?

It would appear then that smart pointers could solve most of our memory management issues, but they were only introduced in the C++11 standard. The C++ standards (*versions*) before then did not have these language features, with older programs being written predominantly in the C++98 or C++03 standard. There remain many such called *legacy* programs which have not upgraded to newer standards, and even then the programmer can choose not to utilise smart pointers in their code.

2.2.1 Research Aim

We therefore propose to create an automatic refactoring tool *(ptr-tidy)*, that performs static analysis on C++ code and aims to refactor usage of raw pointers into smart pointers. It will involve a conservative analysis that first and foremost tries to identify raw pointers that only have a single unique owner at any point in the program. These will be refactored into unique pointers. If this cannot be guaranteed, then a more conservative refactoring to a shared pointer will be applied. We aim to apply automatic refactoring without changing the runtime behaviour compared to the original programs, which we believe will be possible given that C++ language standards are backwards-compatible to a high degree [26].

We use the term refactoring as this is one most familiar to programmers to convey a change in code that does not change its behaviour, but improves its the quality or maintainability. However, our work aligns more closely with the term *source code rejuvenation*, as defined by Pirkelbauer et al. [20]. This is because we exactly aim to replace outdated coding patterns with newer, higher-level abstractions, as well as fitting the notion that this tool would be applied once to a codebase, rather than be a reoccurring task. The two terms can be used interchangeably in the context of this paper, however we will exclusively use the term refactoring for simplicity.

2.3 Research Questions

We outline below the research questions we would like to answer alongside developing the tool. Namely we would like to answer the question of whether such a tool can truly improve C++ code by seeing if the tool can be used outside small examples on larger existing programs, as well as if the refactored results produce code which can be compiled and can be judged to be better. Secondly, if the tool is able to do this, we want to evaluate how well the tool performs by analysing the effectiveness of the analysis algorithms.

RQ1: Can the translation of raw pointers into smart pointers be used to make existing C++ code more modern, safe and extendable?

RQ2: Can the tool reliably identify situations in which unique pointers should be used?

3 Background Survey

In this section we will look into frameworks, design patterns and algorithms that we will use to enable us to read and refactor C++ code and to identify the locations where we can apply our refactorings. We also look into existing memory safe languages, garbage collection as a solution for automated memory management as well as related existing research.

3.1 Clang and LLVM

C++ as a language is enormous, and the first capability our tool will need is to be able to parse (*read*) C++ code. Doing this ourselves would be an undertaking that would surely span longer than the time span of the whole project, therefore it was decided early to use an existing parser. The most capable and mature C++ parsers will naturally be the parsers used by the most popular compilers. This leaves us therefore with a choice between GCC [2] and Clang [3]. The literature [8] is fairly clear on the fact that the design of GCC makes it unsuitable for integration with other projects. Clang on the other hand allows us access to the Clang abstract syntax tree (AST) and parser internals via a standard API such as consumers and visitors.

Clang is a front-end to the LLVM framework [14] and transforms C++ code into the LLVM intermediate representation (IR). The LLVM IR is a language-independent, static single assignment (SSA) [21] representation of the program and will be the target on which we apply our analysis to identify refactoring opportunities. We believe this choice will lead to several benefits in terms of the analysis power and future usage of the project:

[2] `https://gcc.gnu.org/`
[3] `https://clang.llvm.org/`

- First, the usage of SSA allows us to efficiently generate a definition-usage graph, therefore allowing an efficient analysis of all the usages of a pointer to identify its number of owners.
- Secondly, by separating the parsing and analysis, we make our analysis stage reusable by any other LLVM frontends as the analysis is source language agnostic.

Clang was also found to be a good choice of library in terms of usability since documentation and examples are readily found. The AST that it produces is also rich in information and closely resembles the original code, as can be seen in Listing 3.1. Information such as the line and column numbers are included, which makes programmatically rewriting source code easy.

3.2 Pointer Analysis

The main capability of the tool is to be able to identify the maximum number of owners a pointer has at all points of the program execution for all pointers in any given program. If this number for a given raw pointer is one, then we can go about converting all usages of that raw pointer into a unique pointer. If it is more than one then the invariants of a unique pointer cannot be satisfied, and we instead have to use the less performant shared smart pointer. A key intuition, mentioned in subsection 2.1, is that a shared pointer can be substituted for most uses of a raw pointer, making it the default fallback.

A pointer may have more than one owner at a time by copying the pointer to another variable. Effectively the address of the underlying region of memory can be accessed through more than one variable, and any accesses of one variable affect the others. For example, a pointer may be freed using one variable, and this would invalidate the other variable, causing a memory safety violation if we try to use it.

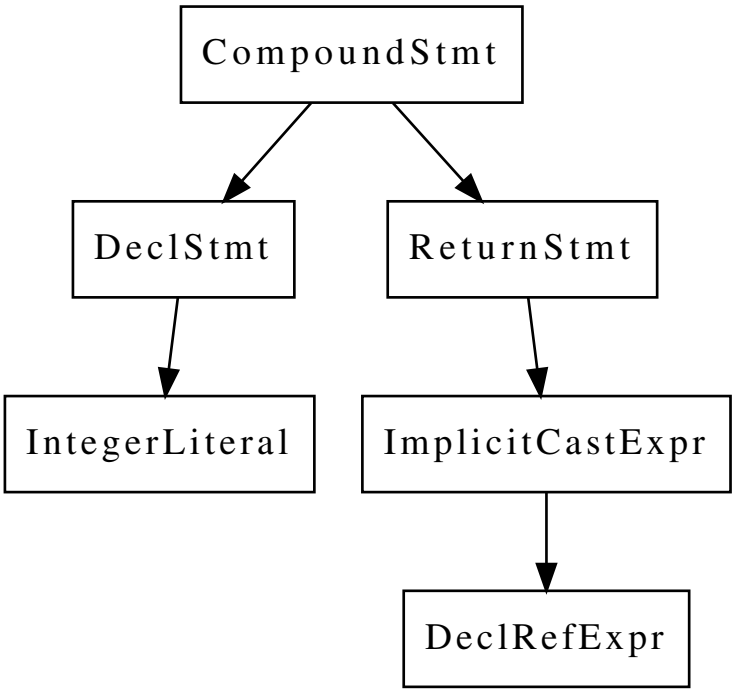
We can formalise the notion of copying a pointer by introducing the concept of *escape analysis*. Escape analysis is used to determine whether an object *escapes* or is accessible from outside the method or thread that created the object [3]. It has been used in languages such as Java [3] to determine whether an object which is created within a method escapes. If it does not, meaning it is only used locally in the method, then it can be allocated on the stack rather than the heap as a performance optimisation. Clang also uses escape analysis for a similar kind of optimisation [4] and also defines the term *pointer capture*. A pointer value is captured if the function makes a copy of any part of the pointer that outlives the call. We can then see that any pointer that escapes must also be captured since there is no way to refer to an object without being able to determine its address, but not every captured pointer also escapes. In Listing 3.2 we can see an example where a pointer may be captured but does not escape and in Listing 3.3 we see an example of an escaped pointer.

We can now see that invariant of the unique pointer are only violated in two conditions:

[4] `https://github.com/llvm/llvm-project/blob/master/llvm/lib/Analysis/CaptureTracking.cpp`

```
int main() {
    int a = 4;
    return a;
}
```

```
TranslationUnitDecl 0x563b7d01f998 <<invalid sloc>> <invalid sloc>
`-FunctionDecl 0x563b7d05f088 </tmp/main.cpp:1:1, line:4:1> line:1:5 main 'int ()'
 `-CompoundStmt 0x563b7d05f2a8 <col:12, line:4:1>
   |-DeclStmt 0x563b7d05f248 <line:2:5, col:14>
   | `-VarDecl 0x563b7d05f1c0 <col:5, col:13> col:9 used a 'int' cinit
   |   `-IntegerLiteral 0x563b7d05f228 <col:13> 'int' 4
   `-ReturnStmt 0x563b7d05f298 <line:3:5, col:12>
    `-ImplicitCastExpr 0x563b7d05f280 <col:12> 'int' <LValueToRValue>
     `-DeclRefExpr 0x563b7d05f260 <col:12> 'int' lvalue Var 0x563b7d05f1c0 'a' 'int'
```



Listing 3.1: Example of converting C++ code to Clang AST. All diagrams show equivalent representation of a small C++ code snippet. The textual representation of the AST shows how information rich each node in the diagrammatic representation is.

```
bool isOdd(int *i) {
    return (unsigned long) i & 1;
}

int main() {
    int *pointer = new int(2);
    return isOdd(pointer);
}
```

Listing 3.2: The pointer is captured in isOdd since the pointer is used for a value that outlives its call, but does not escape since the value of the pointer is not stored anywhere outside the function or thread

```
int *globalPointer;

void escape() {
    int *pointer = new int(0);
    globalPointer = pointer;
    // ...
}

int main() {
    // ...
    escape();
    // ...
}
```

Listing 3.3: The pointer is both captured and escapes since the pointer and the value it points to outlive the call of the function and become accessible by any other method or thread as it is copied to a global variable

- If a pointer escapes, then we do not know where or how the pointer may be used, so we must take a conservative approach and assume the pointer no longer has at most a single unique owner.
- If a pointer is does not escape but is captured more than once, then it cannot be a unique pointer. The case where a pointer is captured exactly once is a special case that we can model as a move of a unique pointer.

3.2.1 Escape Analysis Algorithms

Now that we have defined how escape analysis can be used to identify areas of refactoring, we need to identify a suitable algorithm.

The escape analysis algorithm described by Choi et al. [3] is used as the default escape analysis algorithm in the Oracle Java Virtual Machine implementation [5] and it is widely cited so it forms the basis of our investigation. Due to how similar Java is to C++ and Go in terms of their object-oriented features, it means we can also analyse the escape algorithms used in the Clang and Go compilers.

Choi et al. [3] introduce the abstraction of a *connection graph* which we will see is also used in other implementations. The connection graph captures relationships between heap-allocated objects and object references. The graph is used to perform reachability analysis to determine if an object is local to a thread or method. Most importantly, this abstraction allows for a powerful interprocedural analysis to be performed.

The CG is a directed graph $CG = (N_o \cup N_r, E_p \cup E_d \cup E_f)$ where N_o is the set of objects. N_r is the set of reference nodes which include local variables, parameters, fields and global reference variables. E_p is the set of points-to edges which exist between a reference node that points to object node. E_d is the set of deferred edges, where a deferred edge from one node p to node q signifies it points to what q points to. E_f is the set of of all edges between objects that are a field of another object. We can see in Figure 3.1 how a connection graph would be constructed for simple Java statements.

For each node, we associate a state of either *NoEscape*, *ArgEscape* or *GlobalEscape*. *NoEscape* means that the object does not escape the method in which it was created. *ArgEscape*, with respect to a method, means that the object escapes that method via the method arguments or return value, but does not escape the thread in which it is created. Finally, *GlobalEscape* means that the object is regarded as escaping globally. The initial state for each global node is *GlobalEscape* and *NoEscape* for all other nodes, unless otherwise stated.

The escape analysis is executed through the construction of intraprocedural connection graphs for each method call. We then proceed with creating the interprocedural graph by using the connection graph of the callee to update the connection graph of the caller.

[5] `https://docs.oracle.com/en/java/javase/15/vm/java-hotspot-virtual-machine-performance-enhancements.html`

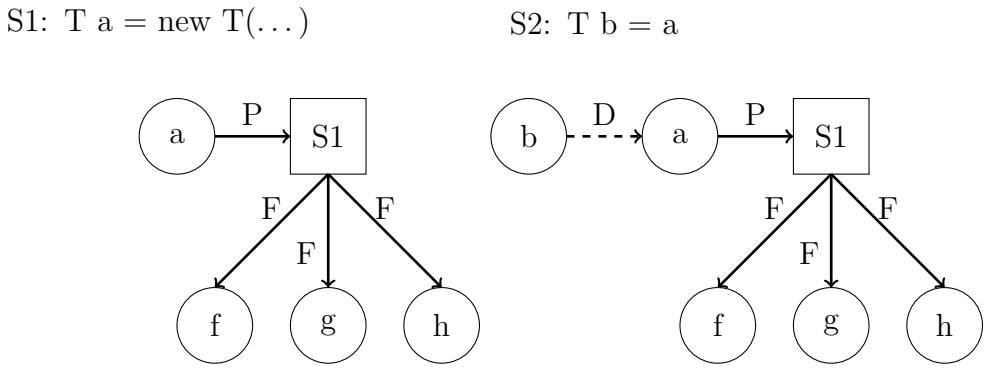


Figure 3.1: A simple connection graph. Boxes indicate object nodes and circles indicate reference nodes (including field reference nodes). Solid edges indicate points-to edge, dashed edges indicate deferred edges, and edges from boxes to circles indicate field edges [3]

At the completion of the escape analysis, all objects which are marked NoEscape and cannot be reached by any node whose state is not *NoEscape* do not escape and are local to the thread in which they are created. We show in subsubsection 4.2.1 how we can adapt this algorithm to check if the invariants defined in subsection 3.2 are met.

Go is another language that uses a escape analysis to allocate objects on the stack rather than the heap. Their escape analysis algorithms is not defined in the specification of the language, however we can look at the source code [6] to determine what algorithm they use. Fortunately the comments give a summary of their algorithm, which involves constructing a weighted graph where vertices represent variables, and edges represent assignments between variables. They walk the graph, looking for assignment paths that violate the invariants of a variable being allocated on the stack and mark those as requiring heap allocation We can see that this approach is highly similar to the use of connection graphs by Choi et al. [3], however there is lacking documentation on how they perform interprocedural analysis so their analysis is perhaps weaker.

Finally, LLVM also uses escape analysis to remove any unnecessary heap allocations. The escape analysis algorithm is also not formally defined, however we can also look at the source code [7]. We see that the `PointerMayBeCaptured` function uses the fact that the single static assignment (SSA) in the LLVM IR allows for efficient creation of definition-usage chains. For any defined value, such as a value initialising a variable, we can traverse all its usages, such as copying to another variable. To determine if an object escapes, we analyse all its usages, and all their usages and so on. For each usage, we then determine if it is an instruction which would cause the object to escape. This leads to a roughly similar analysis as the work by Choi et al. [3] when considering the intraprocedural analysis alone, as we informally explore a graph of the nodes that represent objects and nodes that obtain references to objects by using the original value in a copy instruction. However, its assumption about different instructions are conservative, and it also has no interprocedural

[6] `https://github.com/golang/go/blob/master/src/cmd/compile/internal/gc/escape.go`

[7] `https://github.com/llvm/llvm-project/blob/master/llvm/lib/Analysis/CaptureTracking.cpp`

analysis, making it the weakest analysis so far.

3.3 Memory Safe Languages

There have been many proposed systems languages such as Rust [1], Swift [8], D [9], Nim [10] or Ada (SPARK) [11]. Yet none of these have arguably become mainstream alternatives to C and C++, with Rust and Ada coming closest.

Swift remains mostly a language for application creation in the the Apple MacOS and iOS environments, however it has seen some usage for system programs by companies such as IBM [12]. Ada or its subset language SPARK are used extensively in certain industries to create verifiable programs for critical systems such as in avionics. It is perhaps most well-know for being used in the F-22 fighter aircraft [17], however the use of Ada for those systems show its other weakness in productivity time as it was the cause for several delays in the project, and so it largely remains unused in areas which do not call for such meticulous safety.

Rust has been the most popular language in attracting widespread adoption as a systems language. Its popularity could be quantified by considering that it is being considered for use in writing Linux kernel modules [13], which cannot be said for C++. Following the advice given by many that most systems programs should be written in Rust from now on, we see a potential future usage of our work in the creation of a C++ to Rust transpiler.

Whilst the automatic refactoring of raw pointers into smart pointers solves some memory issues, the language still offers other ways to break memory safety. Equivalent idiomatic Rust would be able to find some of these other errors at compile time. A C++ to Rust transpiler would be a much larger undertaking, however we can envision how our project could be helpful. Rust affine types [1] are semantically equivalent to C++ unique pointers, in that all values have a single unique owner at any point. Therefore our work can identify which pointers are unique and which are not, and those that are not cannot be transpiled into idiomatic Rust.

3.3.1 Garbage Collected Languages

Garbage collected languages encompass some of the most popular programming languages such as Java or Go. They are considered to be simpler languages than C++ since they remove the need for manual memory management by the programmer. In C++, if memory

[8] https://developer.apple.com/swift/

[9] https://dlang.org/

[10] https://nim-lang.org/

[11] https://adacore.com/

[12] https://github.com/ibm-swift

[13] https://lore.kernel.org/lkml/CAKwvOdmuYc8rW_H4aQG4DsJzho=F+djd68fp7mzmBp3-wY--Uw@mail.gmail.com/T

is improperly freed, it gets leaked and can eventually cause out of memory errors. Garbage collectors instead run periodically and scan the memory of the program to find objects in memory which were allocated but are no longer used anymore and then free the leaked objects. Languages such as Java [24] use this fully by allocating almost every object on the heap and not freeing any object, instead relying on the garbage collector to clean up. They are therefore considered to be easier and more productive to use.

Garbage collection does also exist for C++ such as the Boehm-Demers-Weiser conservative garbage collector [2]. So why is garbage collection not used in every language? Garbage collection is not zero-cost as the garbage collection algorithm typically stops the execution of the running program to clean up the memory. This means that latency or pauses are introduced to every program, and can occur at any point in the execution of the program, such as during the hot path of the program.

This has led to garbage collected languages being unsuitable for use in real time systems such as operating systems, as it would be undesirable to have seemingly random latency when interacting with the system. There has been research in modern garbage collection algorithms such as ZGC [15] and Shenandoah [9] for Java which aim to reduce the number and duration of pauses where the garbage collector runs, increasing the responsiveness of the Java virtual machine. However, even with much reduced duration of pauses, it may still be unsuitable for some systems with strict latency requirements.

3.4 Existing Tools and Research

As mentioned in subsection 2.2, our work closely aligns with the research done around *source code rejuvenation*. We believe research in this area is going to increase, as languages move and evolve much quicker than can be supported in enterprise environments, which is supported by the fact that other relevant work has been completed on this subject.

First and foremost, we have evidence that this is a problem that actually exists in industry through a paper by Wright et al. [27] which describes a tool used in Google for running such rejuvenations across a large codebase and shows an example of running a simple rejuvenation of upgrading an API call. They only use Clang for the parsing of code into an AST and traversing the tree to identify and perform simple refactorings. This allows for less complex refactorings, however it also allows for a flexible design to implement different refactorings using the same tool, rather than the focus we have for a single specific refactoring goal.

Secondly, there has been separate academic research into rejuvenation tools such as the work by Hück et al. [12] and Kumar et al. [13]. The former also uses the Clang and AST driven approach that we have described in subsection 3.1 and by Wright et al. [27], while the latter uses a representation called IPR [7], which is a general and efficient data structure for representing C++ programs. We have not evaluated IPR as a representation to use instead of the Clang AST, but we can however comment that whilst we appreciate the aim of the project to create a efficient and compiler-independent representation, the greater

```
int main() {
    int a = 4;
    return a;
}
```

```
...

; Function Attrs: noinline norecurse nounwind optnone
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 4, i32* %a, align 4
  %0 = load i32, i32* %a, align 4
  ret i32 %0
}

...
```

Listing 4.1: The LLVM IR representation of a small C++ code snippet. Listing has been summarized for clarity.

amount of documentation that Clang has, as well as it being the choice of several other successful projects leads us to believe that we will not lose any effectiveness in our use of it.

4 Proposed Approach

4.1 Current Progress

As mentioned in subsection 3.1, we plan on using Clang for parsing C++ code into an AST and LLVM IR for the analysis. We can traverse the AST and look for nodes of interest, such as variable declaration nodes. Given a node in the AST, we can find its equivalent declaration in the IR. This is mainly possible since we can see in Listing 4.1 that the variable names can be matched from the code and therefore AST to the IR.

We can then apply escape analysis for this particular value declaration, which will give us a boolean `true` or `false` result. Based on the returned value, we know if the value does not escape and if so, we can refactor all the uses of this value. This will involve changing the type of the initial declaration, its definition, as well as removing all deletions of the pointer, which can be seen in Listing 2.1 and Listing 2.2. The diagram in Figure 4.1 shows a visual representation of this proposed design.

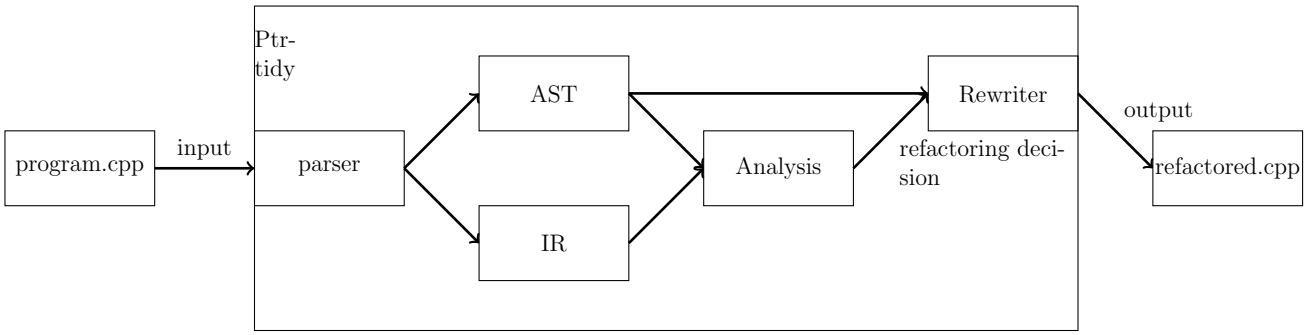


Figure 4.1: Proposed design of Ptr-tidy tool. Parser is the Clang C++ parser which produces an AST and IR representation of the input code. Analysis is the chosen escape analysis algorithm. It produces a decision of whether the pointer is suitable for refactoring, and passes this to the refactoring component which rewrites the code using smart pointers to produce the safer output program.

This design is largely inspired by the componentised design of LLVM [14] and brings several advantages. Firstly the analysis is performed on the IR and so it could be reused by the frontend of a different language and not just necessarily Clang. Secondly this also enables us to flexibly test different algorithms as the C++ specific parsing and refactoring code is not affected by the choice or power of the analysis.

This design has been already used to create a prototype of the tool [14]. The prototype shows that combining the AST and IR is viable and uses the already implemented LLVM escape analysis on variable declarations. It also shows how Clang library allows for easy refactoring since the AST stores all source text and line numbers, making text substitution trivial. Substitution of AST nodes rather than source text substitution has not been explored, however this because Clang documentation states that generally the AST should remain immutable.

The prototype does not currently combine the analysis with the refactoring, which will be the next logical step. The source refactoring is also not fully implemented, and is currently only a proof-of-concept. However, Listing 4.2 shows an example output of running the prototype on a small code block, and it shows how the core idea of the project is achievable.

4.2 Proposed Further Work

As mentioned in subsection 4.1, further work needs to be completed to create a fully capable first version of the tool. This will involve completing the code refactoring component and combining this with the analysis. This first version will allow us to use the tool for initial benchmarking, as described in subsection 4.4, and this will also likely expose unhandled edge-cases, allowing us to refine the design before moving on to completing more complex analyses. This means we can quickly achieve a minimum viable product.

[14] `https://github.com/a-usov/ptr-tidy/tree/c44e67022ba5fa2b3a541f4e730230c1e378e46e`

```
int test(int *b) {
    return 2;
}

int main() {
    int a = 9;
    int b = 2;
    int *c = &b;
    test(&b);
}
```

```
<input.cc:1:10, col:15> Variable b is captured
<input.cc:6:3, col:11> Variable a is not captured
<input.cc:7:3, col:11> Variable b is captured
<input.cc:8:3, col:13> Variable c is not captured

int test(std::shared_ptr<int>b) {
  return 2;
}

int main() {
  int a = 9;
  int b = 2;
  std::shared_ptr<int>c = std::make_shared<int>(b);
  test(&b);
}
```

Listing 4.2: Output from using prototype tool on a small program. We see the analysis of the variables in the code, as well as source code refactoring to use smart pointers. Analysis and refactoring has not yet combined as b should not be refactored since the analysis identifies it as being captured, due to not being able to analyse across function boundaries.

Afterwards, the focus will be on improving the analysis by implementing an interprocedural analysis, as we envision that the existing LLVM analysis will give us poor results. This is because pointers are more often used across function boundaries, as otherwise the programmer could have used a stack variable instead.

4.2.1 Interprocedural Analysis

We propose using a modified version of the escape analysis algorithm by Choi et al. [3] to identify if the proposed invariants in subsection 3.2 are violated. The original algorithm already determines whether a pointer escapes or not, however it cannot determine what the maximum number of unique owners a pointer will have throughout the execution of the program. We can reuse their idea of the connection graph and add an extra condition.

In Figure 3.1 we can see that when we copy a pointer, a new reference node with a deferred edge is created, pointing to the original pointer. Therefore if each reference node has at most one incoming edge, then we know that each pointer is copied at most once, and therefore only transfers ownership once. This therefore means that the value will have at most one unique owner throughout program execution. This relies on careful modeling where using pointers as function arguments is modelled as the creation of new reference nodes, which can be correctly modelled in C++ as it has pass-by-value semantics.

Such an algorithm should give us better results as it will be able to determine whether pointers which are returned from functions escape, as this is a case which the intraprocedural analysis cannot determine and has to take a conservative approach in assuming it escapes.

4.3 Shared Pointer Cycle Analysis

There is an emphasis in the project that the refactored program should not differ in runtime behaviour. The implementation of shared pointers in the C++ standard library however cause a possible issue to this aim. First of all, shared pointers are implemented using reference counting [4], so there is already a performance detriment compared to raw pointers. We choose to omit performance detriments, as they are likely to be marginal, however we will benchmark if any exists.

The issue that we are concerned with are functional requirements of programs, specifically that in some cases, shared pointers will fail to free memory. This occurs when shared pointers are used to create reference cycles, such as can be seen in Listing 4.3. This would go against our aim of improving memory safety.

We therefore propose to have an analysis stage for checking for reference cycles. If we identify a cycle, we do not perform any refactoring and leave the code as it was. To identify cycles, we create a type graph of all the user defined types, and we can use depth first traversal to identify a cycle. We do this by marking visited nodes in the graph as

```
struct Person {
    std::shared_ptr<Person> partner;
};

int main(){
    std::shared_ptr<Person> alice = std::make_shared<Person>();
    std::shared_ptr<Person> bob = std::make_shared<Person>();
    alice->partner = bob;
    bob->partner = alice;
}
```

Listing 4.3: The Person class is used to create a reference cycle. To deallocate the person field in bob, the alice object must be first deallocated. To do so, the bob object in must be deallocated first and so on, meaning the memory will never get deallocated.

visited, and if during the traversal an adjacent node to the current node being analysed is already marked as visited, then a cycle exists.

4.4 Benchmarking

There are several different benchmarks and tests we will perform to analyse the performance and correctness of the tool. Firstly, it was mentioned in subsection 4.3, we will benchmark our refactored programs against the originals for performance, so see if our refactoring efforts cause a significant detriments. Depending on the benchmark program and the effectiveness of the program, the refactoring could only include new unique pointers, which would result in identically performing programs.

Secondly, where automated test exist for the refactored programs, we will use them to ensure that the logic of the program is unchanged after refactoring.

Thirdly, we want to analyse how powerful our analysis is in identifying all opportunities in upgrading pointers to unique pointers. Our tool has to be conservative in some of its assumptions so might not be able to identify all opportunities. We can measure the amount of unique pointers compared to the number of all pointers. However, this number will fluctuate massively depending on what program it is run on, such as one handling resources such as files. Therefore, the method we propose is to measure the performance of our tool against human refactored test cases, to see how much worse our analysis is compared to human effort.

The programs that will be used for the benchmarking will be the subset of C++ programs in the SPEC CPU benchmark suite, as they will usually be pointer heavy, as well as allowing us to measure any possible performance detriments.

5 Work Plan

We have performed a risk analysis for the project, our design as well as our work plan, seen in Figure 5.1, which outlines a schedule for the completion of this project:

5.1 Risk Analysis

- Firstly, the frameworks we have chosen to use, namely LLVM and Clang are mature and enterprise-backed open-source frameworks, and therefore we see no risk in using them for current and any future development,
- Secondly, the work plan has been structured in such a way that given any unforeseen setbacks in the project, our existing prototype can be finished without the integration of more complex analyses to provide a minimum viable product which can be used in experimentation and evaluation. This ensures that we can answer our research questions for this project.

5.2 Gantt Chart

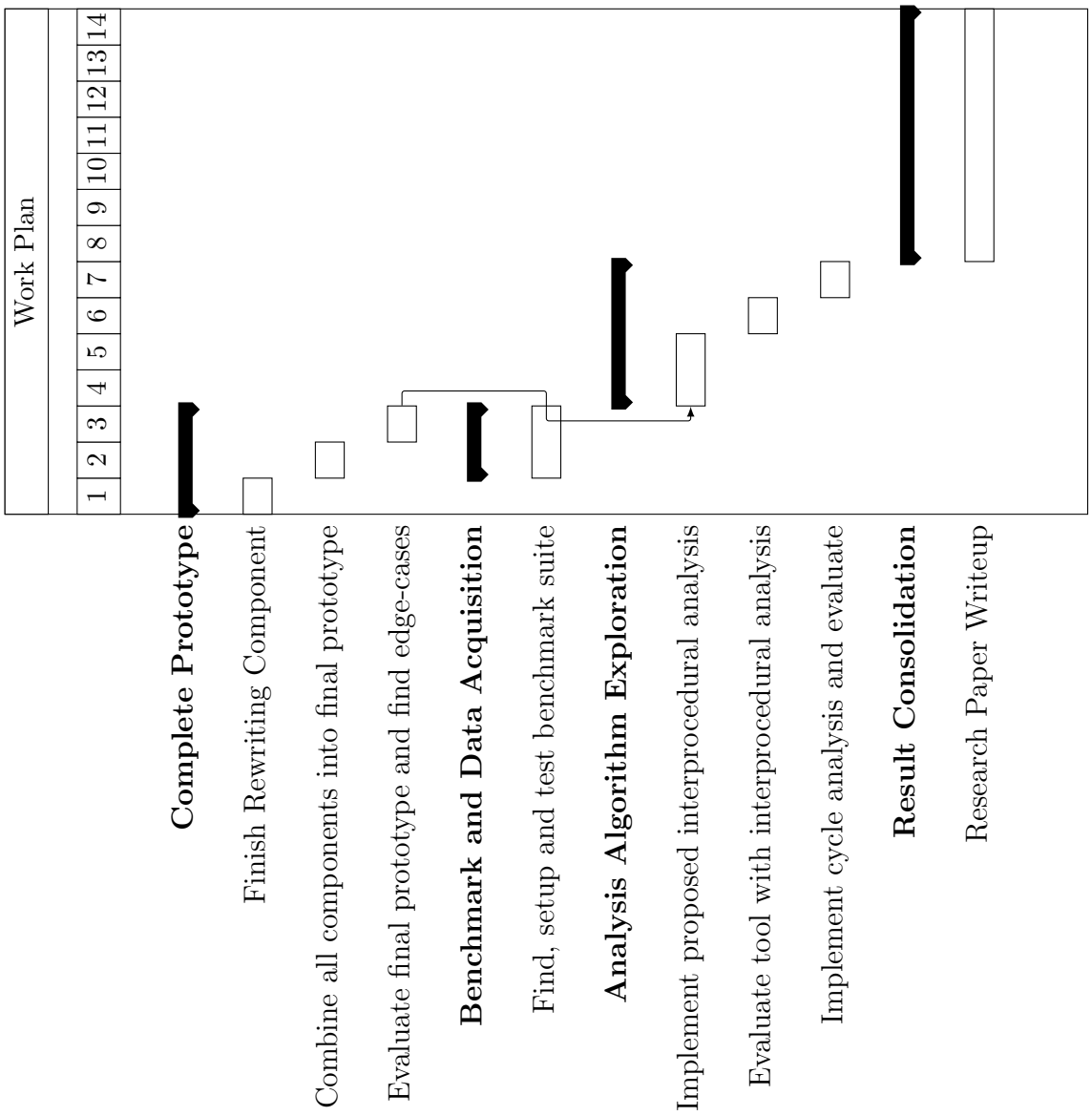


Figure 5.1: A Gantt chart demonstrating the project schedule, broken down over the first 14 weeks of the year from Monday 11th January until Friday 16th April.

References

[1] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari ć, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[2] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for c and c++, 2002.

[3] Jong Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 1999. doi: 10.1145/320385.320386.

[4] George E Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

[5] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 69–80, 2003.

[6] Peter Dimov, Beman Dawes, and Greg Colvin. N1450: A proposal to add general purpose smart pointers to the library technical report. *C++ Standards Committee Papers*, 2003.

[7] Gabriel Dos Reis and Bjarne Stroustrup. A principled, complete, and efficient representation of c++. *Mathematics in Computer Science*, 5(3):335–356, 2011.

[8] Edward B Duffy, Brian A Malloy, and Stephen Schaub. Exploiting the clang ast for analysis of c++ applications. In *Proceedings of the 52nd annual ACM southeast conference*, 2014.

[9] Christine H Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 1–9, 2016.

[10] Alex Gaynor and Geoffrey Thomas. Linux kernel modules in rust. In *Proceedings of the Linux Security Summit North America 2019*, 2019.

[11] Diane Hosfelt. Implications of rewriting a browser component in rust, 2019. URL `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/`.

[12] Alexander Hück, Jean Utke, and Christian Bischof. Source transformation of c++ codes for compatibility with operator overloading. *Procedia Computer Science*, 80: 1485–1496, 2016.

[13] Aditya Kumar, Andrew Sutton, and Bjarne Stroustrup. Rejuvenating c++ programs through demacrofication. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 98–107. IEEE, 2012.

[14] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization*, 2004. doi: 10.1109/CGO.2004.1281665.

[15] Per Liden and Stefan Karlsson. Jep 333: Zgc: A scalable low-latency garbage collector, 2018. URL https://openjdk.java.net/jeps/333.

[16] Yi Lin, Stephen M Blackburn, Antony L Hosking, and Michael Norrish. Rust as a language for high performance gc implementation. *ACM SIGPLAN Notices*, 51(11): 89–98, 2016.

[17] John A Malas. F-22 radar development. In *Proceedings of the IEEE 1997 National Aerospace and Electronics Conference. NAECON 1997*, volume 2, pages 831–839. IEEE, 1997.

[18] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation, 2019. URL https://github.com/microsoft/MSRC-Security-Research/raw/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[19] Information Commissioner's Office. Ico fines british airways £20m for data breach affecting more than 400,000 customers, 2020. URL https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2020/10/ico-fines-british-airways-20m-for-data-breach-affecting-more-than-400-000-customers/.

[20] Peter Pirkelbauer, Damian Dechev, and Bjarne Stroustrup. Source code rejuvenation is not refactoring. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 639–650. Springer, 2010.

[21] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[22] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[23] Bjarne Stroustrup. *A Tour of C++*. Addison-Wesley Professional, 2018.

[24] C Tauro, M Prabhu, and V Saldanha. Cms and g1 collector in java 7 hotspot: Overview, comparisons and performance metrics. *International Journal of Computer Applications*, 43(11), 2012.

[25] Gavin Thomas. A proactive approach to more secure code, 2019. URL https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[26] Titus Winters. P1863r1: Abi - now or never. *C++ Standards Committee Papers*, 2020.

[27] Hyrum K Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. Large-scale automated refactoring using clangmr. In *2013 IEEE International Conference on Software Maintenance*, pages 548–551. IEEE, 2013.