VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Whiley Memory Analyser

Benjamin Russell

Supervisor: David J. Pearce

Submitted in partial fulfilment of the requirements for
Bachelor of Engineering with Honours in Software
Engineering.

Abstract

The Whiley Memory Analyser (WyMA) is a tool for performing a static analysis on Whiley files to evaluate their worst case memory consumption. We evaluate worst case memory consumption to avoid potential errors in systems with limited memory, and it is evaluated statically to ensure it is the absolute worst case. In embedded systems overloading memory can cause unexpected behaviour and hard to diagnose errors, in safety critical systems these issues can cause harm or damages. Whiley already performs static analysis on its code to verify specifications which include type constraints, this means that there were already procedures in place that will help our analysis. We use type constraints to determine the memory requirements of variable declarations, and we use those to analyse the requirements of the programs structures until we know the worst case memory consumption of the file as a whole. Later we used WyMA to assess the memory needs of an example Whiley file to evaluate its accuracy, and see how it can be used to improve the code by guiding refactoring.

Contents

Chapter 1

Introduction

1.1 Motivation

In this project we aim to create a tool which can take a Whiley file and return the worst case memory consumption (WCMC) of the program using static analysis. Knowing the memory requirements of a program is important, especially in the case of embedded systems where available memory is typically more limited, or in safety critical systems where we need to ensure potential errors are mitigated wherever possible. Some example embedded systems are the ATtiny85 with only 512 bytes of RAM [1], and the Arduboy which has 2.5 KB of RAM [2]. When available memory is exhausted it can cause unpredictable or difficult to diagnose errors. The program may throw an error, it may crash the device or it may begin corrupting older data by overwriting it, resulting in erratic behaviour (such as with the ATtiny85). All of these outcomes are problematic, especially in safety critical systems where crashing or behaving unpredictably could endanger people.

Static analysis is the analysis of code without executing it (often performed at compile time) to come to conclusions about any and all execution paths. With static analysis it is possible to guarantee the absence of certain errors. There are many methods and examples for statically analysing code. Some common uses are in definite assignment checking, type checking, checking for unreachable code, and worst case execution time (WCET) analysis. Definite assignment and type checking typically use the abstract syntax tree (AST) to examine the control flow graph of a method at compile time. Both analyse execution paths to categorise values for preventing errors, either from null pointers or invalid casts. Unreachable code checking also analyses the control flow graph at compile time. It sees if any nodes in the AST do not appear in any execution paths so as to warn programmers that the execution might not go as expected.

With the use of our tool we will be able to determine the WCMC of Whiley files so we can know how much stack space is needed to run those programs safely. In the future this could be extended to other languages which would greatly increase the potential scope of its use. Once a tool has this functionality anyone should be able to avoid any and all errors caused by exhaustion of the stack.

1.2 Approach

Our tool will employ static analysis to ensure we get the absolute WCMC rather than the greatest result from testing. Testing cannot guarantee the true WCMC unless it tries all possible combinations of values which is almost never an option.

Whiley is a programming language that uses the Java virtual machine. Whiley is an at-

tempt at a verifying compiler which uses static analysis at compile time to ensure that several common errors are not present, and that user written specifications can be proved true to help establish correctness [3]. WyMA traverses the Whiley file being analysed, recording method declarations, variables and any other syntactic elements that increase the needed stack space. The tool analyses the invariants of the types for these variables to determine their possible range of values to calculate their maximum potential stack needs. Once we have these memory requirements we can display them organized by the containing method. Knowing the containing method and what other methods they call we can determine the memory needs of each method and so, the entire program.

In this project we are focusing on stack memory. We are not considering heap memory because the heap is not normally used in small embedded systems or safety critical systems [4] which are the focus of this project. Stack memory is also easier to analyse in this way. Embedded systems typically have limited memory and computational power. This lack of memory space makes it much easier to overflow resulting in failures. A non-embedded but safety critical system is less likely to run out of memory but the consequences are greater. A common guideline for safety critical coding is to avoid dynamic memory allocation. This is due to unpredictable behaviours associated with it (such as garbage collection or `malloc`), and that the mishandling of memory allocation and deallocation introduces many errors [4].

1.3 Contributions

The goal of this project is to give our users information about their programs memory consumption. We want our users to be able to easily find the maximum potential memory consumption, and to see where in the program large amounts of memory are being demanded so that they could more easily reduce these requirements. We need to provide concrete values for the amount of memory needed and make it very clear where significant usage comes from, especially from unbounded values.

In Chapter 3 we discuss the details of the structure of a Whiley file, and how we handle each data type and each structure for building our assessment of WCMC. In Chapter 4 we go into the details of the implementation of our tool. In Chapter 5 we are evaluating our tool by two major factors: the accuracy of the potential memory consumption evaluation, and clarity about where a file consumes the most memory. The former could be partially evaluated by measuring the memory used in an execution of a file and comparing it to our tools determined needs. We would need to ensure that all of the executions need less memory than our determined bound. This method has the usual limitations of testing. The latter criteria can be evaluated through practical use, by refactoring files based on our tools outputs and seeing how the memory requirements can be improved. We would expect to be able to reduce memory needs from being potentially infinite, and typically even further.

Chapter 2

Background Survey

2.1 Embedded and Safety critical systems

There are many examples of people working on static analysis of programs to determine memory needs as it is an important matter for embedded and safety critical systems. An example of this is AbsInt annotating the control flow graph of files to determine the worst case memory consumption (WCMC) of programs [5]. Safety critical systems are often embedded and embedded systems have more limited memory availability [6]. Embedded systems are difficult and expensive to update after deployment making developers need to ensure as few corrections as possible are required later. Safety critical systems are relied on to be robust, as errors or deviant behaviour in a safety critical system can result in harm or damages.

As memory is often so constrained in embedded systems we can expect errors arising from exceeding this limit to be fairly common. This means there is a demand for ways to avoid this issue, more so due to the significant subset of safety critical systems. In addition to our project making an effort to evaluate the memory requirements of programs there are many others who produced similar systems, some of which are mentioned below.

2.1.1 Coding standards

Coding standards are practices and guidelines used to constrain the use of a language, they are about assessing and mitigating risks, and determining if the remaining risk is acceptable. Standards are often used for safety critical systems to avoid frequent and large problems in these systems to reduce risk of harm or damages.

As an example, a factor of the JSR302 standard (a standard for safety critical applications in Java) is to remove dynamic class loading and constrain dynamic memory allocation which reduces the use of the heap. The benefits of this constraint are that it allows deallocation without reliance on garbage collection, and it reduces memory fragmentation by using sequential scopes forming a stack, both allowing memory space to be more efficiently used.

The Power of 10 rules provide a minimal and easy to remember set of guidelines for making the analysis of safety critical (and often any) systems more reliable, and to accommodate automated compliance checking [4]. These rules include factors such as only using simple control flow structures, keeping the control flow graph (CFG) clear and easy to analyse by not using GOTOs, jumps or recursion. Some more generic rules for safety critical systems include bounding all loops to ensure termination and avoiding heap memory allocation [4].

Another good practice when it is available (which it is in Whiley) is minimizing valid ranges of values to reduce their demands on limited memory.

2.2 Memory usage

Different languages handle memory differently. As an example Java relies heavily on use of the heap, frequently dynamically allocating memory and relying on garbage collection to cull it down. The heavy heap use and unpredictability of garbage collection makes Java generally unsuitable in embedded or safety critical systems without applying coding standards that make it more viable (such as JSR302 which solves this by simply reducing heap use).

In languages like C and C++ much of the memory use can be more explicit in its allocation and deallocation (such as through `malloc`), allowing programmers to be in control of the amount of memory required by a program. This predictability lends itself to embedded and safety critical systems as it increases the awareness and control of memory used.

2.2.1 The Heap

The heap is used for dynamic allocations, to store temporary variables for operators or storing variables that will outlast the method. Dynamic memory allocation is also committed towards the heap for data which the amount of space needed was not determinable at compile time. This data is not able to be allocated part of a stack frame due to the unknown amount needed (such as an array which has memory needs dependent on the instances length). The unpredictable nature of the size of the heap (it being composed of unknown sized items) makes it a hazard to systems with limited memory due to it potentially needing more than is available.

The third Power of 10 rule is to avoid dynamic memory allocations where possible [4]. This is because memory allocators (like `malloc`) and garbage collection have unpredictable behaviours. There is also a significant number of errors introduced by mistakes in handling memory (using deallocated memory, overloading the memory capacity, using memory outside of your allocation, and so on) which can be mitigated by restricting applications to a fixed and preallocated area of memory. Not using heap memory limits applications to the stack.
Java uses the heap heavily, with all objects of user defined types (such as classes) being stored in the heap. A statement as simple as printing a concatenated string will need to dynamically allocate several char arrays to store each stage of the concatenation that does not fit in the prior array, before allocating space for the actual string object to be printed [7].

2.2.2 The Stack

The stack is a more structured form of memory filled by method calls. When a method is executed it puts a stack frame on the stack. These frames contain the address to return to, parameters (argument and return variable locations), current local variable values and a pointer to the parent frame, as shown in Figure 2.1. When there are too many, or too large stack frames, the stack will overflow and cause errors.

The stack is the focus of this project as we are oriented towards safety critical and embedded systems. As the target domain typically avoids heap memory in favour of the stack so do we with our tool. We are organising memory needs by method and taking the maximum demand of all of its local variables to get the size of its stack frame, as well as checking all of its method calls to get the requirements of the child frames (the stack frames placed higher on the stack).

Recursion in safety critical and embedded systems is also avoided for much the same reasons as using the heap. Recursive structures are difficult to bound and can be potentially

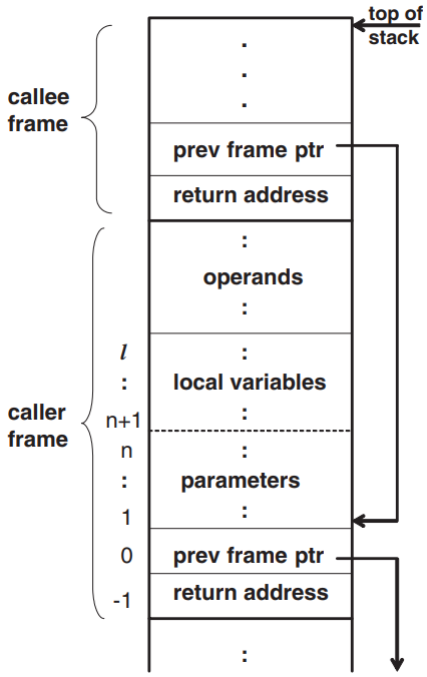

Figure 2.1: Representation of the stack from [8].

infinite (such as `public int x(){return x()+1;}`), and the unknown number of stack frames has the same overflow potential as the unknown heap [4]. Without recursion the stack can be statically derived, allowing compliance with a memory bound to be proved [4]. With this structure our project will be able to simply find a conservative peak in stack usage with the total, rather than the actual peak in memory use. This conservative approach is standard in static analysis, it relaxes absolute accuracy in every case to have a safe result which is valid in all cases.

2.3 Static analysis

Static analysis is needed if we want the true WCMC. Static analysis is the examination of a program without running it. Some common forms of static analysis are definite assignment, unreachable code, and type checking at compile time in many languages such as Java, as opposed to dynamic testing such as unit testing performed at run time. Static analysis allows for judgements based on all possible values while dynamic testing uses specific combinations of values (which is very rarely exhaustive) and produces what judgements it can. Testing based approaches may miss entire executable paths and will also not see the full results of concurrency [6], for example, if two threads peak at different rates it is likely for testing to finish before they ever occur at the same time giving an underestimation of requirements. Because of these distinctions, static analysis is needed to ensure the true WCMC is found as it can find the absolute maximum needs in memory space for these programs.

Static analysis does have its limitations where dynamic testing has the advantage. Static analysis is typically used to find certain types of errors (such as definite assignment and so on) at compile time rather than having them occur at runtime. There are many errors which may appear at runtime which cannot be practically found with static analysis due to the need to decide between false positives and false negatives. For example when dealing with

definite assignment we need any potential errors to be handled, but in another situation it may only be viable to identify the definite errors.

With static analysis we can prove software correctness by writing formal specifications to be used with a theorem prover (such as with Whiley or Dafny). These specifications can be used to assert properties of values and the relationships between them. This could be used to say that for example variable `A` is equal to 1, variable `B` is greater than variable `A` or that a function returns a value less than its parameter. Test cases derive from the specifications, but often they are much easier to write than a formal specification to describe the complete behaviour being specified.

2.3.1 Worst Case Memory Consumption

Albert et al talk about how taking the total usage of the heap is a safe way to statically judge how much memory is needed, but will typically overestimate due to deallocations [9]. This result is still valid, an execution will never exceed the resultant value, making it a safe result. Their work studied garbage collected languages and used Java as an example. In their system they were dealing with garbage collection which is fairly unpredictable in its timing and will only happen at run time, so they took measurements looking for where the peaks in uncollectable memory occur. It is in an earlier publication that they show their work on determining what data is collectable [10]. Albert et al were aiming to use this to infer bounds on heap consumption accounting for deallocation which would give them a more accurate WCMC.

AbsInt say that many static analysis methods will overestimate requirements depending on what stage the analysis is performed at as there can be later optimizations [5]. Because of this, they argue that bounding memory needs requires a low level language (or functional language, or the intermediary forms closer to execution for a higher level language) to be analysed for reliable results [8]. As when looking for WCMC we want an upper bound, an overestimation can still be a valid result so for this we can ignore the effects of optimization. Working with the code of a higher level language such as Whiley has advantages of its own; being readable makes it easier to work on, and they avoid some challenges such as indirect branching.

2.3.2 Worst Case Execution Time

WCET is a common form of static analysis that is used to determine the worst case execution time of a program. There are a variety of methods for analysing WCMC statically; converting code into a series of functions [10], bounding needs from execution paths [6, 8], or bounding individual statements from the control flow [5, 7, 11].

In addition to existing memory analysis techniques there are processes for WCET which have similar behaviour to our WCMC analysis. WCET analysis is a static analysis used for determining the maximum needed time to confirm that it can complete execution within a hard threshold on specific hardware for real time computing. This can be taken as a parallel to WCMC, as WCMC analysis is used to find the maximum needed memory to ensure an execution can fit within a strictly limited memory space. With this project the intention is similar to Ferdinand et al's, as it is also to determine requirements of statements and use them collectively to produce a complete evaluation [5].

The method used by AbsInt for evaluating WCMC was much the same as their method for WCET where they assign values to each statement in the control flow graph and annotate it, giving where it peaks as the WCET [5]. The work done by Puffitsch et al was also based

on methods for finding WCET, totalling values assigned to statements [7]. Puffitsch et al's system is however less sophisticated than the one produced by Ferdinand et al, and needed user annotations to aid it amongst other issues.

2.4 Whiley

Whiley is a programming language created by Dr David Pearce that uses the Java Virtual Machine to execute. Whiley features the use of extended static checking to ensure the absence of certain errors, this checking is done as part of the automated theorem proving that is used to verify formal specifications written into the code [3]. Being able to include formal specifications in your code means that you can often ensure correctness in a program through preconditions, postconditions and other invariants.

The Whiley compiler (WyC) verifies programs at compile time to ensure user given specifications are correct and that several common errors (null pointers and divide by zero for example) do not occur. The WyC does not however ensure termination or bounded memory use, especially as integers are unbound. For Whiley, there is a description of a system for extracting the ranges of values for integers from the specifications in the program. The specifications used include type invariants, loop invariants and pre and postconditions [11]. The system used is available as a prototype in the Whiley2EmbeddedC (WyEC) package which includes operators for ranges, allowing us to process the sets of possible values [11].

In Whiley, integer values are normally unbounded (making them potentially infinite and so a serious problem for when memory is limited) but the user written specifications can restrain it and other variables to clearly defined ranges [11]. For example;

```
type boundint is (int x) where 0<=x && x<16
type boundarray is (boundint[] x) where |x|<10
```

In this example a `boundint` is an integer restricted to the range of 0 to 15 which only requires 4 bits to store, rather than the standard integer ranging from - ∞ to ∞ and requiring potentially infinite memory. Other than basic variables (that is to say one object of one type) which will have memory needs based on type and constraints, some interesting variables we will measure are arrays, which need space for the contained type multiplied by the length of the array (which can also be infinite), and records, which need the sum of the requirements of contained objects.

The `boundarray` uses `boundint`s and its length has a range of 0 (as it cannot have a negative length) to 9. Because `boundarray` has a limited length and we know the size of `boundint`s we know that a `boundarray` requires 36 bits (4 multiplied by 9) for its elements.

Whiley is open source (available at `https://github.com/Whiley`) including several other projects built on Whiley which can be useful for this project. As Whiley already performs static analysis to verify correctness in specifications we can use the existing procedures for this to assist us. We can use the procedures for traversing the file (such as `AbstractVisitor`) to find all of the variables and structural statements, and the procedures for validating the specifications can be used by us to read the invariants for our own processing, and the work on integer ranges can be used in our evaluation of integer based types [11].

Chapter 3

Design

Whiley can constrain variables with type invariants and other invariants with their use. It treats many collections as primitives and is intended for static analysis. All of these aspects help with the aims of this project. These aspects assist us by making the memory requirements for variables clearer and meaning there are well established methods for analysing Whiley files we can use. This assistance means that this project is much simpler for working with Whiley than it would be with other languages. The Whiley compiler also aids us by giving us access to ways of compiling Whiley files and for handling AST's. These aspects are fundamental to WyMA as explained below.

As the specifications of this project include that we are working on Whiley files there are many packages and files that are relevant, and available work which we would functionally have to duplicate if we were not to use them.

3.1 Whiley Abstract Syntax Tree

Having access to the Whiley compiler (WyC) means we have access to the Abstract Syntax Tree (AST) it produces, giving us a simple way of processing the file. The AST lets us traverse the file in a processed form, so we could handle the labelled statements without needing to scan or parse them ourselves.

The Whiley AST contains the nested structure of elements that make a Whiley file. By traversing the AST we can find every statement and every variable we need to allocate memory for. By finding them through this structure we will also have the trace up the tree, declaring all of the parenting statements. In Figure 3.1 we show an example Whiley file which uses many syntactic elements of Whiley, and in Figure 3.2 we show half of the AST of the file in Figure 3.1. Figure 3.2 shows the structure of the type declarations for `boundint` and `boundrecord` but not for `boundunion` or `boundarray`, it also shows the function `func` but not the method `main`. The visible structure of ASTs will allow us to know for example, if a variable is global, in a method, in an if statement in a method and so on. As shown in Figure 3.2's use of `p.a` (the lowest node in that representation highlighted in the red rectangle), we know that in Figure 3.1 `a` is from a record which is accessed in a return statement, in the true branch of an if/else statement, which is in the body of the `func` function.

Using this structure assists with our goal of helping users see where in the file there is significant memory usage. This tree allows us to inform users of which statements cause significant memory use, and which scope it occurs in so they can find them more easily.

Using the Whiley AST has the advantage of being already implemented and simple to

```
type boundint is (int x) where 0<=x && x<16
type boundunion is (boundint|null x)
type boundarray is (boundunion[] x) where |x|<=4
type boundrecord is ({boundint a, boundarray b} rec) where a<4

method main():
    boundint i = 0
    while i<3:
        func({
            a: i,
            b: [1,2,3,null]
        })
        i = i+1

function func(boundrecord p) -> (boundint r):
    boundunion b = p.b[a]
    if b is null:
        return p.a
    else:
        return b
```

Figure 3.1: An example Whiley file that uses a variety of syntactic elements.

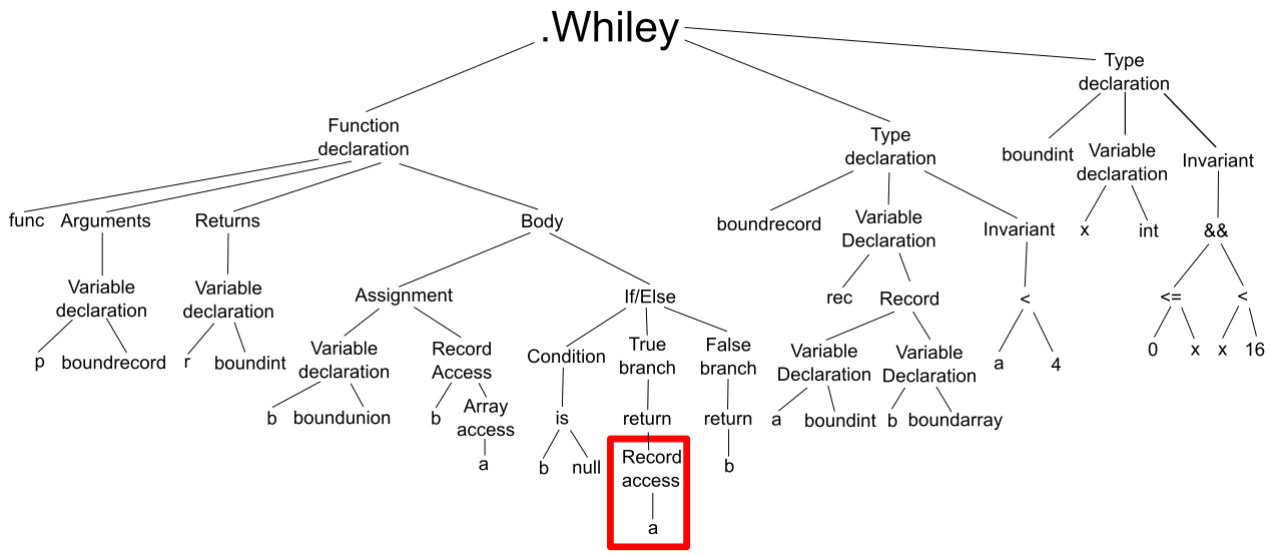

Figure 3.2: A subset of the AST for the code in Figure 3.1.

integrate. If we did not use them then we would need to do our own parsing (functionally making our own AST) to be able to recognise this structure. Also in the AST under the type declarations there is an invariant tree as shown in Figure 3.2, these subtrees can be processed to determine a valid range of values for a type, as will be discussed later.

There are no distinct disadvantages to using the AST, except for not knowing about later optimisations like we would if we were analysing the resulting machine code [5]. The machine codes would be less informative of the specific needs of objects than the invariants in the AST, and performing our analysis before optimisation will typically result in a higher bound which is safer.

3.2 Memory Analysis

Many syntactic elements are based on the sub-elements they contain, with the full meaning of most nodes in the AST being based on its children. For example in Figure 3.2 the type declarations are composed of a name, a variable and constraints, most of which have their own children. WyMA will need to evaluate the memory requirement of the base statements (the leaves of the AST) and properly collate them as it gets closer to the root, representing the file as a whole. The following gives an overview of some common AST nodes.

- The most generally used structural element is a function or a method. As they are collections of statements they will need space for everything their execution requires, going through all contained statements individually.

  In Figure 3.1 the method `main` needs space for the variable declaration of `i` and the while loop (described below) it contains, and the function `func` needs space for its argument, its return value, the variable declaration of `b`, and an if/else block (described below).

- A loop is a self contained scope, all declarations contained in the loop being deallocated at the end of each cycle, so a loop should never need more memory than is necessary for one cycle. The loop itself does not require memory, but in our model it does act as a container for its children to be passed up the tree.

- Conditionals such as if/else statements contain disjoint sets of statements, so as we look for the maximum this statement requires memory for the greater set but not both. As an example, `if(bool){20 bits}else{10 bits}` would require 20 bits as the branches require 20 (x)or 10 bits, and 20 is the safer amount.

- Method or function invocations are where another function or method is performed, and this invoked function or method will have its own stack frame. To have the memory needs of a function or method accurately represented, it needs its own frame and every frame above it included in WyMA's assessment of it. To do this we treat the invocation expression as having the memory requirements of the invoked function or method.

  This reflects how an invocation can jump across the control flow graph. Performing the invoked function or methods execution before the declarations of the calling function or method can be deallocated. In Figure 3.1 when `main` calls `func` it needs to allocate space for the invoked method in a stack space above itself. If there is no room for `func` then `main` would not be able to execute successfully.

  The issue with this model is how recursion would be handled. As the needs of an element are based on its children if a node descends from itself it will get into an infinite

```
type boundint is (int x) where 0<=x && x<16
type boundunion is (boundint|null x)
type boundarray is (boundunion[] x) where |x|<=4
type boundrecord is ({boundint a, boundarray b} rec) where a<4

method m():
    T x = ...
```

Figure 3.3: A copy of the type declarations from Figure 3.1 and a stub of a variable declaration in a method.

loop and never resolve. As determining if it terminates is a large and separate problem and recursion is discouraged in safety critical systems [4] we decided to have any recursion be treated as potentially infinite. Having recursion be identified as potentially infinite is safe in that it will be greater than or equal to the number of actual recursive cycles, as we cannot determine the depth of the recursion. The downside of this method is that any shallow recursions are caught in this approach and are considered infinite.

These ways of handling structures are designed to assist our goals by improving accuracy, reducing evaluated memory where possible while ensuring we do not evaluate below the maximum possible to ensure our result is correct.

There are many more statements which cause memory allocations (operators, array generators/constructors, etc) which get allocated to the heap, often only to be assigned into a stack variable before deallocation. As these are heap elements which should not be used in our targeted systems [4] we will not handle them precisely. We should instead throw an error to let users know when a case is not being handled so as to not mislead with our outputs.

3.3 Bit Width Calculation

WyMA needs to evaluate the maximum number of bits needed by each variable. Finding the memory space needed by a variable declaration depends on the type it is based on. When a user defined type is instantiated its bit width is determined by the base type but may be further constrained by the invariants.

In the example in Figure 3.3 the method `m`'s memory needs are based on variable `x`, which is in turn based on the specification of type `T`. In Figure 3.3 there are a few such types; `boundint`, `boundunion`, `boundarray` and `boundrecord`, which are based on an `int`, a union, an array and a record respectively, with all but `boundunion` being constrained further. Below are descriptions for how several common types are assessed.

- A boolean value can either contain false or true, represented as 0 or 1 and so only requires 1 bit to store.
- The amount of memory needed by a byte value is simply 8 bits.
- An int value is normally unbound, needing potentially infinite memory for potentially infinite values. If invariants are used to constrain it, then the bit width is determined by the range of valid values, needing a number of bits that can hold the upper and lower bounds specified.

In Figure 3.3 for `boundint` there is a valid range of values from 0 to 15, and so, 4 bits are required.

- `void` and `null` cannot contain values and so do not need any memory allocated for them on their own.
- Arrays are fixed length collections of variables of a common type, which would need the sum of memory for the contained values. The factors in this are the possible values being stored (being defined by their type) and the length of the array. Array length is normally any natural number (integers from 0 to ∞) but can be constrained by invariants. The bit width of an array is the product of the bit width of the contained type and the maximum length of the array, being potentially infinite if either of those aspects are.

  Using `boundarray` from Figure 3.3, the range of valid array lengths is from 0 to 4, as an empty arrays content fills 0 bits, the greater potential need is for 4 `boundunion` objects. A `boundarray` needs 23 bits to store all of them; 5 bits for each of the 4 `boundunion`'s (as explained below), and 3 bits to store the length of the array.
- Records are collections of variables, so their memory needs are the sum of the contained variables. As shown in Figure 3.3, records can also apply further invariants to their fields. In this case restricting the `boundint` in `boundrecord` to be less than 4 and so only actually needing 2 bits, which with the 23 bits for the `boundarray` means that `boundrecord` needs 25 bits.
- A union type is a variable that can be any of a set of types and so needs to be able to fit any of them, simply having a bit width of the greatest of its types. A union will also need a few bits to act as a flag for which type it is currently. The `boundunion` in Figure 3.3 needs 5 bits; 1 to flag if it is a `null` or a `boundint` and 4 bits to store a `boundint` (those 4 also being able to contain the 0 needed for `null`).

3.3.1 Type constraints

The Whiley2EmbeddedC (WyEC) package [11] (which includes the work on integer range analysis for Whiley) will be directly applicable to our project. Integers in Whiley are potentially infinite so for us to get a non-infinite and accurate result for required memory space we need to constrain these values. We use the constraints in type declarations such as the `boundint` from Figure 3.3 and process their invariants to get a range of valid values. We can extract bit widths from each range to determine the total memory needs for integer based values (such as integers and array lengths). We will not look at a methods pre and post conditions for further constraints as they are only enforced at the start and end of the method and not during the body where the variable could be assigned a much greater value. We also do not include the variables used in formal specifications as they are never instantiated.

Relying on type constraints for our tool demands that only custom types and bound primitives be used in the program for it to be non-infinite. In our targeted systems you would want variables to be as constrained as possible or as needed, with an unbound value being a risk, making this an acceptable limitation on use.

3.4 Architecture

The structure and rough control flow of WyMA is shown in Figure 3.4. When WyMA is executing the first stage is that we either receive a .whiley or a .wyil file (the two entry

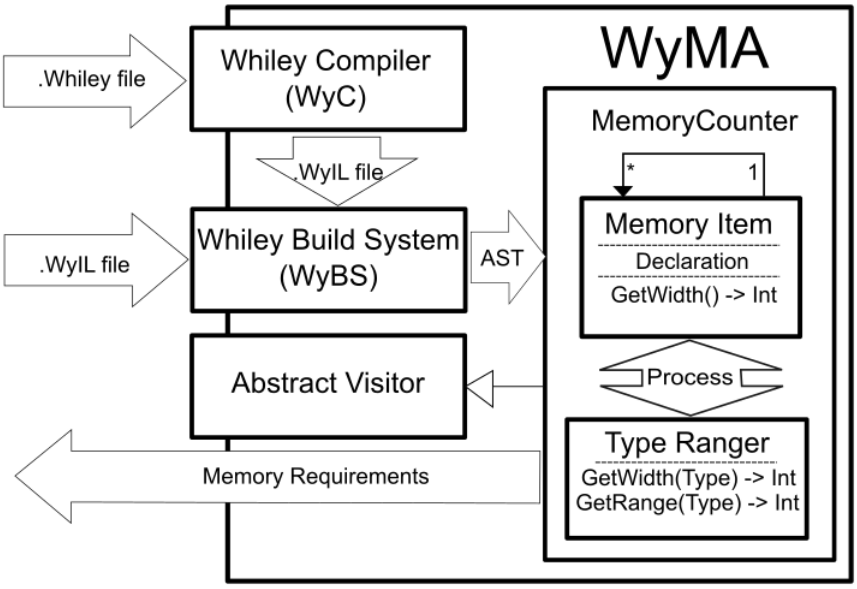

Figure 3.4: Architecture of WyMA.

points in Figure 3.4), if it is a .whiley file we compile it into a .wyil to simplify extraction of the AST. We then extract the AST from the .wyil file and begin our analysis (the entry into MemoryCounter in Figure 3.4). We navigate the AST with our `AbstractVisitor` to visit all reachable nodes in the tree, to analyse them in a depth first traversal. For each relevant node in the tree we process its bit width, either by recursively handling its children (as described in Section 3.2) or if it is a variable, by processing its type (as described in Section 3.3). Once our traversal and analysis are complete we return the information about the WCMC we have determined.

WyMA is dependent on a few existing Whiley systems. It uses WyC to produce WyIL files. This dependency is largely optional, WyMA could still function without it by only taking already compiled files. Allowing compilation as part of analysing the memory usage saves users some effort, and speeds up the cycle of refactoring for reducing memory usage. WyMA also needs access to the Whiley build system (WyBS) to be able to extract ASTs, the importance of which was described in Section 3.1. Thirdly is the dependency on `AbstractVisitor` (also from WyC), which facilitates traversal of the AST.

Chapter 4

Implementation

Chapter 3 described the structure of WyMA (such as the control flow in Section 3.4), and the details of the required behaviour (Sections 3.2 and 3.3). In this chapter we present the details of how WyMA implements these requirements. We discuss how WyMA compiles Whiley files and extracts its AST, how we traverse the AST and produce our memory structures, and how we evaluate the memory needs of variables in our leaf nodes. At the end of this chapter we discuss some of the limitations of WyMA; the types and structures we do not properly handle, and how optimisation in compilation may change the memory needs to differ from WyMA's assessment.

4.1 Compiling Whiley

When WyMA is executed it expects an argument that is the file location of either a .whiley or a .wyil file. If the file received is a .whiley then it will be compiled using the functionality from the Whiley compiler (WyC). If compilation is successful then it will produce a .wyil file in the same location as the .whiley was found. Having the compiled file be put in the same place with the same name allows for an easy transition into using the new file, it also makes it easy for the user to find afterwards. If the compilation fails, then we throw an error so we do not try to process an invalid file.

This compilation includes the Whiley standard library (v0.2.3) to allow access to a wider range of common elements. Processing a file that uses other libraries would be difficult as there is no way in WyMA to look at arbitrary libraries.

4.2 Produce Whiley AST

The first task our tool always performs is to read the provided .wyil file and produce a Whiley AST from it. We produce the needed path from the argument given (if a .whiley file was given the only difference will be the file extension). From the path we create a Whiley build system (WyBS) project and read the .wyil, decoding it into an AST.

4.3 Initial Traversal

We used the `AbstractVisitor` from WyC to find the AST nodes relevant to our analysis. The `AbstractVisitor` traverses all items in the AST, from methods and their parameters to the constants in an assignment statement. We extended the `AbstractVisitor` to perform additional functions when certain nodes are found; variable declarations, function or

method declarations, if/else statements and invocation expressions. These extensions produce nodes for our own parallel tree, a simplified AST missing the irrelevant nodes. The nodes of the simplified tree are implemented to support the recursive functions needed for our analysis. During the traversal we kept a stack of nodes that resembled the path from the root to the current node, this allowed us to easily determine which higher node is the parent of any given node.

We created our simplified tree to cement our understanding of the structure, how to interact with it, and to simplify the later traversals for refining the analysis and printing the representation for the user. The full functionality could however probably be achieved in the first traversal as our tree is mostly a stripped down version of the original AST.

4.4 Control Flow

We set up our tree nodes such that it recursively processed the tree items. All of the stored items are either typed variables, or their needs are based on their children. The default case for determining a nodes width is to check if it is a variable and if so use its types width, else it sums the child nodes widths. This default case applies to variables, loops, functions and methods which complies with the requirements described in Section 3.2.

The exception cases are in invocations, conditionals, and the file as a whole. For conditionals the width was overridden by that of the greatest child (following Section 3.2), and instead of having all of what the `AbstractVisitor` finds under it as its children, we deliberately separated the branches. For function or method invocations, children are handled normally but processing its width is different. Our invocation processing does not look for the invoked method until after the tree is completed. When the width of an invocation is being processed it performs a depth first search of the tree, also storing any functions or methods found. If it finds a cycle in the control flow graph it will return potentially infinite due to the recursion, and if it finds the invoked method it will use the requirements of that. The file as a whole acts as an invocation of the `main` method (which is a simpler search as it will be a direct child of the root), summing all of children if `main` is not found.

That we treat recursion as infinite because of the cycling in control flow being potentially infinite is supported by Holzmann [4], stating that recursion should not be included in safety critical systems because termination cannot be proved. Refusing recursion is also supported as Whiley does not ensure termination in its programs, and to prove that a recursive structure terminates is beyond the scope of this project.

4.5 Bit Widths

When processing variables for their bit widths, we do so through their types in a combination of the base type and its invariants. WyMA caches the bit widths it finds for a type in the current program so it does not need to process it again if it is used multiple times as it is likely to. We start by caching a variety of primitives (integer, boolean, byte, null) to be able to handle user defined types based on them, and we build up the cache as types get used in the program.

As types are introduced we determine the bit width needed by them, often based on the range of valid values for them, which we can determine from the type invariants. We extract the invariants from the Whiley files using `WhileyFileResolver` from WyC, this retrieves the type declaration from which we can take the invariants. Invariants are made up of trees of usually binary and unary expressions. The expressions at the leaves are variables and constants with a range of valid values, as you traverse up the tree the operators join the

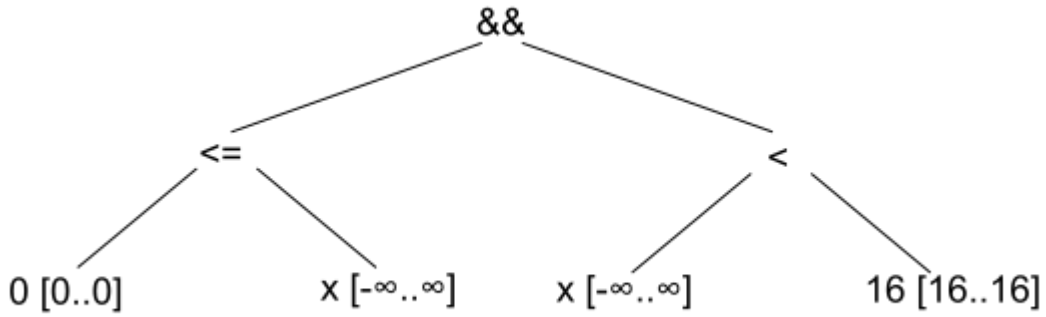

Figure 4.1: Invariant tree of boundint from Figure 3.1

ranges until at the root there is a range of valid values for the variable the invariant applies to.

```
type boundint is (int x) where 0<=x && x<16
```

Above is the definition of `boundint` from Figure 3.1, and Figure 4.1 shows its invariant tree. The constants only have one valid value at all times (the value of the constant) and variables at the leaves can be anything their type allows, in this case as `x` is an integer and integers are unbound in Whiley they have a range of -∞ to ∞. Following the left side of Figure 4.1 we have the <= comparator which produces two ranges as an output: one for how the left hand side changes and one for the right, in this case it will result in [0,0] and [0,∞]. In this case [0,∞] will be passed up to the `&&` operator as WyMA can identify which sides are relevant to the current variable. If both sides of the expression are relevant, it will use the intersection of the results to maintain that both remain correct.

The various types get treated as specified in terms of Section 3.3, the ranges some of them are based on is managed in much the same way as the previous example in processing invariants. In invariants we had to manage a variety of expressions: mathematical operators which translate the range (and sometimes need translated back afterwards), logical operators that require treating the ranges as sets, comparators that produce two output ranges, and the various variable access expressions (such as accessing a record or getting the length of an array).

Records are some of the more complicated and interesting cases which demanded different solutions be developed. The key feature of records is that they have multiple variables, and each of these variables can have its own distinct invariants. For example:

```
type r is ({
    int a,
    int b
} c)
where c.a >= 6 && c.b <= 9
where c.a <= 12
```

In this example we need both of the invariants on `a` to apply and the invariant on `b` to be separate. Applying multiple invariants is simple, we just needed to `and` them together to get the intersection of ranges. For the latter challenge we needed to have our invariant processing be variable specific, on each branch of the invariant tree one of the objects it passes

```
COMPILING: 028_flag_A.whiley
RESULT: SUCCESS
RESULT:

New type: flag
flag - int[0,4]
New type: flagPointer
flagPointer - int[0,4]

Global - (18)
        partition - (15)
                cols (flag - [3])
                ncols (flag - [3])
                lo (flagPointer - [3])
                mid (flagPointer - [3])
                hi (flagPointer - [3])
                While - (0)
                        If - (0)
                                True - (0)
                                False - (0)
                                        If - (0)
                                                True - (0)
                                                False - (0)
        main - (18)
                colors (flag - [3])
                partition called - (15)
```

Figure 4.2: Output for the refactored 028flag.whiley code (code is in Figure A.3, description in Section 5.1)

up as it is processing is if that branch contains the relevant variable. Knowing where the relevant variables are allows us to determine that `a` has a range of [6,∞] and that `b` has a range of [-∞, 9] instead of getting a range of [6,9] for the record (from the first invariant).

For arrays that our bit width is based on type and length does not account for invariants affecting specific values in the array, but managing that as well would be unnecessarily complicated, and the effect it has can only decrease memory consumption making it a safe simplification.

4.6 Output

The main output of WyMA is the number of bits needed by the program being analysed, but we can also output a string containing our memory item tree. When the tree is printed you can see the entire hierarchy of nodes through indentation, clearly labelled with names and what they are, for easy navigation and identification. All nodes in this string are also labelled with their memory needs to allow users to understand different elements contributions easily.

Figure 4.2 is an example of one of WyMA's outputs, in this case for the `028_flag.whiley` file seen in Appendix A and analysed in Chapter 5. In it you can see at the top that it successfully compiled, in the next block you can see it determining ranges for the user defined types, and the third block is the assessment based on the AST. Figure 4.2 shows the nested structure; such as the nested conditionals inside the while loop, the various variables defined in the function `partition`, and the invocation of `partition` in `main`. You can also see how every node in this AST is labelled with its name so it can be identified, and the number of bits it is estimated to need.

4.7 Discussion

4.7.1 Unhandled cases

There are several cases which WyMA does not yet handle properly, described below.

- Switch statements provide a series of cases, each containing a block. If a case's condition is met its body is executed. If the blocks execution does not include a `break` or a `return` statement, then at the end it will 'fall through' and execute the next case. If no case condition is met (or it is fallen into) the default case is executed.

  WyMA doesn't perform any control flow analysis and handles switch statements simply. WyMA will ignore the switch statement itself but will detect all of the contained nodes, this results in switches being reasoned about as one large block of statements. This reasoning is equivalent to falling through all of the cases, which acts as the worst possible execution of the switch in terms of memory consumption and execution time. This is a safe way of handling switches as it will typically over estimate, but it could be made more accurate.

- Open records are records which contain "`...`" as their last field, this indicates that an instance of that record type will contain at least the specified fields. As WyMA bases its assessment on static types this ambiguity cannot be properly handled as it is. To be able to properly handle an open record, WyMA would need to evaluate them individually when an instance is created, and track which variables reference them across execution. The ambiguity in an open records size means it would be allocated into the heap, which means they should not be used in our targeted systems [4].

- Reference types ("`&`"+Type) act as pointers to values of the given type, which can be dereferenced using an `*.` `new` expressions produce a reference to an object of a given type. The amount of memory needed to store a reference varies based on the processor being used. As WyMA has no way of knowing such details there is no good way for it to handle references.

- Lambda expressions create references to anonymous functions. The static type of a lambda contains only that it is a function, as well as the types of its arguments and return values. The bodies of a lambda can contain invocations and variables, which we cannot know purely from the static type but do need memory allocations. These factors are the same as the reason that WyMA cannot properly handle an open record, and are why lambdas are not currently handled.

4.7.2 Optimisations

Because we perform our analysis on the AST we do not account for any optimisations made by the compiler (as stated in Section 3.1), which may affect the accuracy of the analysis [5]. Optimisations are typically performed in the translation to machine code by the compiler and may be different depending on the compiler used. That these changes are compiler driven means that the effects they would have on the memory consumption would be more difficult to find the cause of (or to change) in the code to improve it. The following are a few examples of optimisations and how they would affect the memory consumption of a program.

- Registers are a quickly accessible section of a processors memory. When generating machine code we would want our compiler to maximise the use of these registers

(rather than the slower to access memory) to improve the performance of execution. If an argument is passed as a register then the called function or method will not need to allocate more space for them, reducing the amount of memory actually needed.

- Functions invoked may be inlined during compilation. Inlining a function copies the function body in place of the invocation. Inlining removes the overhead of calling the function such as creating the stack frame, typically removing the need to allocate space for the functions arguments and return values as they would be declared in the calling frame. This removal of the redundant allocations would reduce the actual memory needs.

- Reading and writing to memory is more efficient when the data is aligned, preferring memory addresses which are a multiple of a certain size. A compiler may pad data to round up to the nearest multiple of that amount to fit into this alignment. This padding assists the performance in terms of execution time, at the cost of consuming more memory. With padding, objects which do not fit into the alignment will be assigned more memory than they actually need, with very small objects being assigned much more than they need. As WyMA finds the maximum memory a variable could actually use, padding will often increase the allocation above this assessment.

Chapter 5

Evaluation

We performed our evaluation of WyMA through case studies. To evaluate the performance of WyMA we refactored the given Whiley files to assess through two metrics. First was seeing how much we could reduce the memory needs of the program while preserving functionality using WyMA as a guide. Second was the accuracy of WyMA's outputs, comparing to our own assessments of the files done by hand.

5.1 Case Study: 028flag.whiley

`028_flag.whiley` is one of the WyBench (the benchmark suite for Whiley [12]) test programs. We used `028_flag.whiley` to evaluate WyMA by seeing how we could refactor the file to improve memory consumption (from potentially infinite), and check the accuracy of the bounded results. `028_flag.whiley` contains a simple program which solves the Dutch national flag problem by assigning colours numerical values so they can be ordered and then sorting the list. The code in the Whiley files and the corresponding outputs are given in Appendix A. Figure A.1 contains the original file and Figure A.3 contains the refactored version, with Figures A.2 and A.4 containing their respective outputs.

5.1.1 Refactoring improvements

The WyBench programs were written without memory consumption as a concern, and so they typically need potentially infinite memory. The aim of our refactoring is to preserve the original code, but to also constrain the memory consumption of the program. In our refactoring of `028_flag.whiley` there were only three significant changes made.

- The first change was that where three global int variables are defined (`RED`, `WHITE`, and `BLUE`) we constrained their constant values. We defined an integer based type which could minimally accommodate the values of these variables and used that to define them. This constraint means that we can use our new type for any reference to these values, needing 2 bits rather than potentially infinite.

  Before

```
int RED = 0
int WHITE = 1
int BLUE = 2
```

  After

```
type col is (int x) where 0<=x && x<=2
col RED = 0
col WHITE = 1
col BLUE = 2
```

- The second change was the removal of `String`s. `String`s are unbound (and so potentially infinite) arrays of characters which needs to be bound or removed. In the case of `028_flag.whiley` the `main` method takes the argument `ascii::string[] args`. `args` is not used at any point and Whiley does not require that argument in the main method so it was removed.

  Before

```
public method main(ascii::string[] args):
    int[] colors = [WHITE,RED,BLUE,WHITE]
    //
    colors = partition(colors)
    //
    io::println(colors)
```

  After

```
public method main():
    flag colors = [WHITE,RED,BLUE,WHITE]
    //
    colors = partition(colors)
    //
    //io::println(colors)
```

- The third change was to bound the `colours` array which is frequently used. Originally this was an unbound array of `int`, which makes it a potentially infinite list of potentially infinite objects. Our solution to this was to define our own array type which bounds both of these dimensions for use throughout the file. To fully make use of this constraint we also changed all of the `nat` and `int` values used to index the array to our own type which is bound on both ends.

  The intention of the program is to be able to process an array of arbitrary length. An array of any length is not viable because of memory limits, the array length needs to be constrained to fit into the capacity. This constraint would generally be based on the largest value you expect to handle or on the possible capacity of the system. We used 4 for this example as it was the length of the only used input (`colors`).

  Before

```
...
    nat lo = 0
    nat mid = 0
    int hi = |cols|
...
    int[] colors = [WHITE,RED,BLUE,WHITE]
```

  After

```
type flag is (col[] x) where |x| <=4
type flagPointer is (int x) where x>=0 && x<=4
...
    flagPointer lo = 0
    flagPointer mid = 0
    flagPointer hi = |cols|
...
    flag colors = [WHITE,RED,BLUE,WHITE]
```

All of these changes are to reduce a potentially infinite aspect to be a bound one. As shown in the outputs (Figures A.2 and A.4) these changes have reduced WyMA's evaluation of `028_flag.whileys` memory needs from potentially infinite to 18 bits (but it should actually require 42 bits as explained in Section 5.1.2). These changes were mostly making bounds explicit and have minimal impact on the programs behaviour. The behaviour altering changes we made were in commenting out a print statement, and in removing `args`. In the case of `028_flag.whiley` the removal of `args` had no change to behaviour but in other programs where such an input is needed you could apply bounds to it the same as any of the other unbound types. The other behavioural change was in limiting the size of the array which could be processed. This limitation is acceptable as in actual use this restriction would be set to the requirements of the target system and the actual use of the program (such as length of used arrays), minimising the impact of the change. These changes could be safely applied to almost any Whiley file.

5.1.2 Accuracy

Figure 5.1 shows the output of WyMA for the refactored `028_flag.whiley`. WyMA made the following evaluations as shown in the brackets after each tree node (for the original code all values outside of the while loop were potentially infinite as shown in Figure A.2):

- `col` needs 2 bits.

  ```
  type col is (int x) where 0<=x && x<=2
  ```

  `col` is an `int` from 0 to 2, so 2 bits is correct.

- `flagPointer` needs 3 bits.

  ```
  type flagPointer is (int x) where x>=0 && x<=4
  ```

  `flagPointer` is also an `int` like `col` but is from 0 to 4 so 3 bits is correct.

- `flag` needs 3 bits.

  ```
  type flag is (col[] x) where |x| <=4
  ```

  `flag` is an array of `col` with a length from 0 to 4. The 0 to 4 range means there should be 3 bits just for storing the length, and another 8 bits to store 4 `cols`. WyMA's assessment of `flag` is incorrect and invalid as 3 is less than the actually needed 11.

- `partition` needs 15 bits.
  The code for `partition` is in Figure 5.2. `partition` needs space for two `flags`, as its argument and return. It also needs three `flagPointers` to store `lo`, `mid`, and `hi`

```
Global - (18)
        partition - (15)
                cols (flag - [3])
                ncols (flag - [3])
                lo (flagPointer - [3])
                mid (flagPointer - [3])
                hi (flagPointer - [3])
                While - (0)
                        If - (0)
                                True - (0)
                                False - (0)
                                        If - (0)
                                                True - (0)
                                                False - (0)
        main - (18)
                colors (flag - [3])
                partition called - (15)
```

Figure 5.1: The section of the refactored 028flag.whiley files output which shows memory use of tree nodes. Complete output is in Figure A.4

as declared at the start of the function. Within the while loop there are no variable declarations, method invocations or anything else needing more memory assigned. All of the leaf nodes within the while are assignment statements between already allocated objects. The memory needs of `partition` is simply that for two `flag`s and three `flagPointer`s, this should be 31 bits but is incorrectly assessed as 15 due to the carried error from `flag`.

- `main` needs 18 bits.
  The code for `main` is in Figure 5.2. `main` needs space for a `flag` and an invocation of `partition`. The invocation should require the 31 bits of the function being invoked, as there needs to be space in the stack for the called functions frame. `main` should require 42 bits but was incorrectly assessed by WyMA as 18 bits, again due to the carried error from `flag`.

5.1.3 Output

Figure 5.1 shows the output of WyMAs assessment of `028_flag.whiley`. As indentation indicates depth in the AST we can see that the top level within the file contains `partition` and `main`. As `Global` (the name used for the file as a whole), `main`, and `partition`s entries all only contain their names and the number of bits needed, we know that they are body nodes in the AST and act as containers for their children.

The `main` node has two children; `colors` and `partition called`. `colors`' entry states a type with its bitwidth indicating that it is a variable declaration of type (and memory needs of) `flag`. `main`s other child is in the form Name + " called - (" + int + ")" so we know it represents an invocation of `partition` and will have the memory needs of the invoked function (or method).

The `partition` node has six children, the first five are variable declarations in the same form as `colors` from `main` (two of type `flag`, and three of type `flagPointer`).

`partitions` sixth child is another body node, a while statement, within this while statement (and its children) there are no nodes that require a memory allocation. Within the while statement there are two nested if/else statements, given the shape of the structure it could be an if/else-if/else but we cannot be sure just from the output in Figure 5.1.

The body nodes correctly determine their memory needs based on their children (correct in regards to the child nodes, not in regards to actual needs), including the invocation which was able to find and use the memory needs of `partition`. In this example the files memory needs are found to be 18 bits, this is because WyMA knows to look for `main` and use its memory needs when possible as `main` will be the base of the stack.

When we look at Figure 5.2 for the relevant part of the code Figure 5.1 was based on, we can see that our observations are correct but with a few interesting points.

- Assignment statements, while numerous, are not included in WyMA's analysis as they do not allocate memory on their own.
- As explained in Section 3.3.1, invariants (and any declarations there in) are not included. This is because they only enforce properties at the start and end of their scope, rather than throughout. They are not executed at run time, so any variables declared for them are not part of execution and will not be allocated memory.

  ```
  ensures all { k in 1..|ncols| | ncols[k-1] <= ncols[k] }:
  ```

  If invariants were included then the variable `k` in the above (from the full refactored code in Appedix A.3) would introduce an unbound `int` to the analysis. This would make WyMA assess as needing potentially infinite memory when it is not actually needing any more.
- Variables for arguments and return values are not treated any differently than the other declarations.

```
function partition(flag cols) -> (flag ncols):
                        //Invariants removed for space
    flagPointer lo = 0
    flagPointer mid = 0
    flagPointer hi = |cols|
    ncols = cols
    while mid < hi:
                        //Invariants removed for space
        if ncols[mid] == RED:
            ncols[mid] = ncols[lo]
            ncols[lo] = RED
            lo = lo + 1
            mid = mid + 1
        else if ncols[mid] == BLUE:
            hi = hi - 1
            ncols[mid] = ncols[hi]
            ncols[hi] = BLUE
        else:
            mid = mid + 1
    return ncols

public method main():
    flag colors = [WHITE,RED,BLUE,WHITE]
    colors = partition(colors)
```

Figure 5.2: The section of the refactored 028flag.whiley files code which produces tree nodes. Complete code is in Figure A.3

Chapter 6

Conclusion and Future Work

6.1 Conclusions

The goal of this project was to create a tool for statically analysing Whiley programs to determine their WCMC. We wanted WyMA to be able to produce results that were as accurate as possible, and to give users enough information about the programs memory use to guide refactoring for improvements. WyMA is an easy to use tool, simply taking a .whiley or .wyil file location and producing an evaluation of its WCMC. With a bit of work, WyMA could likely be integrated with the other common static analysis tasks such as type checking. This integration (or simply WyMA's use) would allow programmers working on embedded or safety critical systems to easily monitor their programs memory consumption to ensure it is acceptable.

As described in Section 5.1.1, WyMA has successfully been used to guide the refactoring of `028_flag.whiley`. As Figure A.2 shows, any types, variables, or methods which use high or potentially infinite amounts of memory are clearly and specifically identified. This identification allows users to easily know where and what needs to be constrained further. Section 5.1.2 discusses the accuracy of WyMA. In this example we can see that WyMA can correctly handle the body nodes representing the control flow and structure of the program, it also shows how WyMA can accurately determine the valid ranges of `int` based types. In Section 5.1.2 we do however see that WyMA currently incorrectly handles arrays, and in Section 4.7 we discussed other situations which negatively affect WyMA's accuracy.

6.2 Future work

6.2.1 Improvements

If development of WyMA were to be continued there are a few immediately apparent things to work on:

- The most obvious task is to deal with the accuracy issues found in Section 5.1.2, and potentially find ways to handle some of the cases in Section 4.7.1.

- Second is the removal of our intermediary step, the simplified AST produced in our initial traversal as described in Section 4.3. This simplified AST assisted in our understanding of the structure and simplified the development of WyMA. As WyMA is largely functional and the simplified AST is mostly redundant, it would be more efficient if WyMA was refactored to perform its analysis on the complete AST. This

refactoring could be done by changing our extension of `AbstractVisitor` to directly call for the nodes analysis, instead of having it produce a parallel node which does the same thing.

- The third task is to thoroughly test the results of WyMA. An interpreter (similar to a compiler but instead of translating code into a different language it executes it) could be implemented to measure memory use at run time. This interpreter could be used to measure files, methods, and functions actual memory consumption for a variety of inputs, and compare these results to WyMA's output for that body node. The execution of these tests would be simple enough to automate, and we would know that there are problems with WyMA's accuracy if a test ever exceed its assessment.

6.2.2 Uses and next steps

A way WyMA could be extended is to also try to perform an analysis of the heap to determine the absolute memory consumption. This should not be needed for embedded or safety critical systems as the heap should be avoided in them where possible [4]. This analysis would be quite different from the stack analysis as it would need to include a greater consideration of object lifetimes and the control flow of the program.

Once the accuracy issues are resolved WyMA could be used by anyone concerned with memory consumption in a Whiley program. The most obvious way to significantly increase the usefulness of our WCMC analysing tool would be to expand on the languages it can be applied to. At first this would likely be other languages which include formal specifications such as Dafny. To include these other languages would require a way of producing an AST of the same format, and a way of translating type constraints into the same form as well. These requirements could be met by having a system to convert the other language's AST and type constraints into Whileys form, or by adjusting WyMA to accept something more generic (which would likely require the conversion of both).

After the languages with formal specifications are accepted we may extend to languages without them. For the AST this shouldn't require much more than for the languages with formal specifications, but the type constraints pose an issue. One option for managing type constraints would be to extend the language to have annotations signifying how types should be used, but enforcing them non-invasively is a larger problem making these languages not very practical to include.

Bibliography

[1] "Attiny85 - 8-bit avr microcontrollers - microcontrollers and processors, https://www.microchip.com/wwwproducts/en/attiny85," accessed 2018.

[2] "Arduboy, https://arduboy.com/," accessed 2018.

[3] "Whiley — overview, http://whiley.org/about/overview/," accessed 2018.

[4] G. J. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[5] C. Ferdinand, R. Heckmann, and B. Franzen, "Static memory and timing analysis of embedded systems code," in *Proceedings of VVSS2007-3rd European Symposium on Verification and Validation of Software Systems, 23rd of March*, pp. 07–04, 2007.

[6] J. Regehr, A. Reid, and K. Webb, "Eliminating stack overflow by abstract interpretation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 4, pp. 751–778, 2005.

[7] W. Puffitsch, B. Huber, and M. Schoeberl, "Worst-case analysis of heap allocations," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pp. 464–478, Springer, 2010.

[8] W.-N. Chin, H. H. Nguyen, C. Popeea, and S. Qin, "Analysing memory resource bounds for low-level programs," in *Proceedings of the 7th international symposium on Memory management*, pp. 151–160, ACM, 2008.

[9] E. Albert, S. Genaim, and M. Gómez-Zamalloa, "Heap space analysis for garbage collected languages," *Science of Computer Programming*, vol. 78, no. 9, pp. 1427–1448, 2013.

[10] E. Albert, S. Genaim, and M. Gomez-Zamalloa, "Heap space analysis for java bytecode," in *Proceedings of the 6th international symposium on Memory management*, pp. 105–116, ACM, 2007.

[11] D. J. Pearce, "Integer range analysis for whiley on embedded systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on*, pp. 26–33, IEEE, 2015.

[12] "Wybench/src at develop - whiley/wybench - github, https://github.com/whiley/wybench/tree/develop/src," accessed 2018.

Appendix A

028flag.whiley

```
import std::ascii
import std::io

type nat is (int x) where x >= 0

int RED = 0
int WHITE = 1
int BLUE = 2

type Color is (int n)
where RED <= n && n <= BLUE

property matches(int[] items, int from, int to, int item)
where all { i in from .. to | items[i] == item }

function partition(Color[] cols) -> (Color[] ncols)
requires |cols| > 0
ensures |ncols| == |cols|
ensures all { k in 1..|ncols| | ncols[k-1] <= ncols[k] }:
    nat lo = 0
    nat mid = 0
    int hi = |cols|
    ncols = cols
    while mid < hi
    where |cols| == |ncols|
    where lo <= mid && hi <= |cols|
    where matches(ncols,0,lo,RED)
    where matches(ncols,lo,mid,WHITE)
    where matches(ncols,hi,|ncols|,BLUE):
        if ncols[mid] == RED:
            ncols[mid] = ncols[lo]
            ncols[lo] = RED
            lo = lo + 1
            mid = mid + 1
        else if ncols[mid] == BLUE:
            hi = hi - 1
            ncols[mid] = ncols[hi]
            ncols[hi] = BLUE
        else:
            mid = mid + 1
    return ncols

public method main(ascii::string[] args):
    int[] colors = [WHITE,RED,BLUE,WHITE]
    colors = partition(colors)
    io::println(colors)
```

Figure A.1: 028flag.whiley original code (comments removed).

```
COMPILING: 028_flag.whiley
RESULT: SUCCESS
RESULT:

New type: ascii::string
ascii::string - int[-inf,+inf]
New type: Color
Color - int[0,2]
New type: nat
nat - int[0,+inf]

Global - (Infinite)
        partition - (Infinite)
                cols (Color[] - [Infinite])
                ncols (Color[] - [Infinite])
                lo (nat - [Infinite])
                mid (nat - [Infinite])
                hi (int - [Infinite])
                While - (0)
                        If - (0)
                                True - (0)
                                False - (0)
                                        If - (0)
                                                True - (0)
                                                False - (0)
        main - (Infinite)
                args (ascii::string[] - [Infinite])
                colors (int[] - [Infinite])
                partition called - (Infinite)
                io::println called - (0)
```

Figure A.2: 028flag.whiley original output.

```
import std::ascii
import std::io

type nat is (int x) where x >= 0

type col is (int x) where 0<=x && x<=2
col RED = 0
col WHITE = 1
col BLUE = 2

type flag is (col[] x) where |x| <=4
type flagPointer is (int x) where x>=0 && x<=4

type Color is (col n)
where RED <= n && n <= BLUE

property matches(flag items, flagPointer from, flagPointer to, col item)
where all { i in from .. to | items[i] == item }

function partition(flag cols) -> (flag ncols)
requires |cols| > 0
ensures |ncols| == |cols|
ensures all { k in 1..|ncols| | ncols[k-1] <= ncols[k] }:
    flagPointer lo = 0
    flagPointer mid = 0
    flagPointer hi = |cols|
    ncols = cols
    while mid < hi
    where |cols| == |ncols|
    where lo <= mid && hi <= |cols|
    where matches(ncols,0,lo,RED)
    where matches(ncols,lo,mid,WHITE)
    where matches(ncols,hi,|ncols|,BLUE):
        if ncols[mid] == RED:
            ncols[mid] = ncols[lo]
            ncols[lo] = RED
            lo = lo + 1
            mid = mid + 1
        else if ncols[mid] == BLUE:
            hi = hi - 1
            ncols[mid] = ncols[hi]
            ncols[hi] = BLUE
        else:
            mid = mid + 1
    return ncols

public method main():
    flag colors = [WHITE,RED,BLUE,WHITE]
    colors = partition(colors)
```

Figure A.3: 028flag.whiley refactored code (comments removed).

```
COMPILING: 028_flag_A.whiley
RESULT: SUCCESS
RESULT:

New type: flag
flag - int[0,4]
New type: flagPointer
flagPointer - int[0,4]

Global - (18)
        partition - (15)
                cols (flag - [3])
                ncols (flag - [3])
                lo (flagPointer - [3])
                mid (flagPointer - [3])
                hi (flagPointer - [3])
                While - (0)
                        If - (0)
                                True - (0)
                                False - (0)
                                        If - (0)
                                                True - (0)
                                                False - (0)
        main - (18)
                colors (flag - [3])
                partition called - (15)
```

Figure A.4: 028flag.whiley refactored output.