Removing The Stack For Fun And Profit

(Implementation of Stackless C)

Kris Nobes

Bachelor of Science in Computer Science with Honours
The University of Bath
April 2008

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:

Removing The Stack For Fun And Profit

(Implementation of Stackless C)

Submitted by: Kris Nobes

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).
This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

This project investigates techniques for minimising the use of the stack in C applications, via source-to-source transformations. Transformations are suggested and implemented for two aspects of C: heap allocation of program data (variables) and control flow information (return addresses). Variables can be stored in heap allocated structures for each scope to achieve “data stacklessness”. Addresses of labels and inline unconditional jumps can be inserted for “control flow stacklessness”.

The effectiveness of the transformations is evaluated and suggestions made to simplify manipulation of a parse tree for source-to-source systems which operate at the tree level. It is shown that through the proof of concept implementation that this approach is possible, but considerably harder than anticipated. Discussion revolves around the design choices made, the reasoning behind them and how this project may be continued through future work and improvements. A new question and problem is posed to further this line of research.

Contents

1 Introduction **1**

1.1 Aim . . . 2

1.2 Objectives . . . 2

1.3 Document structure . . . 3

2 Literature Review **5**

2.1 Stacks . . . 6

2.1.1 What is a stack? . . . 6

2.1.2 Why are stacks used? . . . 8

2.1.3 How are stacks used? . . . 9

2.2 Stacklessness . . . 11

2.2.1 What is it to be stackless? . . . 11

2.2.2 Why stackless? . . . 12

2.2.3 Techniques of implementing the stackless language . . . 12

2.3 Uses . . . 13

2.3.1 Security . . . 13

2.3.2 Threading & Debugging . . . 17

2.3.3 Development of transformation and source-to-source systems . . . 18

2.4 Existing systems . . . 18

2.4.1 Type safety . . . 18

2.4.2 Concurrent programming . . . 20

2.4.3 Translation & rewriting systems . . . 21

2.4.4 Tools & frameworks . . . 22

2.5 Summary 24

3 Technology Analysis 25

3.1 Compiler frontend 25

3.1.1 Lexical analysis 26

3.1.2 Syntactic analysis 26

3.1.3 Semantic analysis 26

3.1.4 Transformation 27

3.1.5 Code generation 27

3.2 Parsing 27

3.2.1 Grammars 28

3.2.2 Parsing algorithms 29

3.2.3 YACC/GNU Bison & Lex/Flex 30

3.3 Program transformation 31

3.4 C 32

3.4.1 Scopes & variable declaration keywords 32

3.4.2 Problems 33

3.5 Summary 34

4 Proposed Solution 36

4.1 Requirements 36

4.1.1 Functional requirements 36

4.1.2 Interface & software requirements 37

4.1.3 Non-functional requirements 37

4.1.4 Data stackless requirements 37

4.1.5 Control flow stackless requirements 37

4.2 Modelling the stack 38

4.2.1 Separation of control flow & program data 39

4.3 Options for heap allocation of data 40

4.3.1 Options for heap allocation of program data 40

4.3.2 Options for heap allocation of control flow information 41

4.4 Implementation choices . . . 43
4.4.1 A fixed stack size? . . . 43
4.4.2 Contiguous or fragmented stack? . . . 44
4.4.3 Garbage collection or free()? . . . 44
4.4.4 Binary tree or n-ary tree? . . . 44
4.4.5 Singly or doubly-linked tree? . . . 45
4.4.6 Compiler usage . . . 45
4.5 Transformations required . . . 45
4.5.1 Problems & limitations . . . 46
4.6 Summary . . . 49

5 Implementation 50
5.1 Tree generation - lexing & parsing . . . 50
5.2 Data stackless method . . . 51
5.2.1 Environments & semantic analysis . . . 51
5.2.2 Structure construction . . . 52
5.2.3 The stack . . . 53
5.2.4 Loading & storing values . . . 57
5.2.5 Function calls . . . 57
5.2.6 Function declarations . . . 60
5.2.7 Global variables & initialisation . . . 61
5.3 Control flow stackless method . . . 62
5.3.1 Inline assembly . . . 62
5.3.2 Transformations . . . 63
5.4 Other functionality . . . 65

6 Discussion 66
6.1 Results . . . 66
6.1.1 Analysis . . . 66
6.2 Critical evaluation . . . 67
6.2.1 Portability . . . 67

6.2.2 Language choice . . . 67
6.2.3 Success criteria . . . 68
6.2.4 Success against criteria . . . 68
6.2.5 Limitations . . . 69
6.3 Project continuation . . . 70
6.3.1 Potential improvements . . . 70
6.3.2 Improving call stackless transformation . . . 71
6.3.3 Alternative choices . . . 72
6.3.4 Future developments . . . 73
6.4 Tree manipulation framework . . . 73
6.5 Summary . . . 75

7 Conclusion **76**

8 Bibliography **78**

A Code **83**
A.1 Results . . . 83
A.1.1 Input code . . . 83
A.1.2 Post data stackless transformation . . . 84
A.1.3 Post data + call stackless transformation . . . 86
A.2 Source code . . . 89
A.2.1 Compilation & execution notes . . . 89
A.2.2 Source code excerpts . . . 89

List of Figures

2.1 Program memory 7

4.1 Modelling the call & data stack on the heap 39

5.1 Mapping of environment structures to data frames 56

6.1 Inserting a statement into tree 74

6.2 Inserting a statement into tree 74

List of Listings

3.1 Identifier/typename ambiguity example 34
4.1 Storage of heap-allocated stack 39
4.2 Extraction of EIP value 41
4.3 Example of change of semantics for global scope 47
5.1 Structures produced for a function definition 52
5.2 Example use of **void***∗* in data frame 53
5.3 Initialisation of heap-allocated stack 54
5.4 Original and transformed code setting up state shown in figure 5.1 54
5.5 Cast required to access structures within data frames on data sub-stack . . 57
5.6 Transformation of function call 59
5.7 Transformation of function call and return value 59
5.8 Transformation of function declarations 61
5.9 Function call post-data stackless transformation 63
5.10 Function call post-data and call stackless transformation 64
A.1 Input test code 83
A.2 Data stackless transformed code 84
A.3 Data + call stackless transformed code 86
A.4 Source code excerpts from main transformation (c2c.c) 89
A.5 Environment header (environment.h) 104

Acknowledgements

Many thanks to my supervisor, Professor John ffitch and Martin Brain. Both provided their time and knowledge, suggestions and support, for which I am grateful. The knowledge gained and progress made on this project would not have been possible without their intellectual contribution. I acknowledge and appreciate the efforts of friends and family who have contributed suggestions and moral support to both this project and myself. Thanks.

Chapter 1

Introduction

This project seeks to investigate the implementation of stackless C, producing a tool which provides an effective way of transforming standard C source code to a stackless version. More accurately, since it is not possible to entirely remove the use of the stack using transformations at the source code level, the aim is therefore to minimise the use of the stack.

The main modern imperative languages in use today use the program stack to store activation records in a contiguous block of memory as a core aspect of their execution. Function calls require a call stack to store the return address i.e. the location of the next instruction to execute once the called function has finished executing. A location to store data when passing parameters to a function and storing local variables (when recursion is permitted) is also needed. A solution is the use of activation records, which are generally used to pass data to a function being called. The required data is stored on a single stack, typically consisting of local variables, actual parameters and a return address [1].

C is an example of a programming language which uses the program stack in this manner. Stack based data storage is both a fast and simple method of storing and passing data to achieve the desired effect (compared to heap allocation). There are however, a number of problems and disadvantages to using the stack in this way - some of which can be minimised or avoided using a variety of techniques.

Security problems arise through buffer overrun attacks (such as "smashing the stack"[2]), which exploits the combination of control flow and program data in the same structure. Stack overflow can occur (particularly on Windows systems) when the stack data exceeds the memory available for the program stack (which requires a block of contiguous memory since the stack is comparable to an array).

C has been modified for various reasons before. There exist safe dialects of C such as Cyclone from AT&T Research which extends the language as well as helping to guard against some of the problems associated with C (e.g. dangling pointers) [3]. CCured is a C source to C source translator which inserts memory checks.

A stackless C would circumvent a number of problems with memory limits and enables various methods to defend against buffer overrun attacks to be implemented. As well as helping to alleviate a number of security exploits, the implementation of stackless C would allow a number of "limits" imposed on programs to be lifted. co-operative threading could also be implemented - while very rarely the main style of process scheduling in operating systems nowadays (with a few exceptions, mostly for earlier versions of today's OSs, real-time systems and embedded OSs), this would be very useful in a number of applications.

Stackless C can be considered to be an experimental project that shall deliver a system which can hopefully achieve a number of the objectives established. While the general area being covered is well-known (but not well discussed), the project has not been done before and therefore has an element of research to it since a variety of techniques are investigated throughout the project. Delivery of the project and dissertation will require more than just a basic implementation of a reasonably simple and well defined system capable of achieving a common set of tasks. Instead, implementation techniques for transformations, source-to-source rewriting and tree manipulation systems must also be considered.

Finally, our natural curiosity impels us to investigate and learn more about that which interests us but isn't necessarily well known.

Why C? C was chosen since is it the most portable abstraction of what computers actually do, as well as being a widely used, popular and powerful programming language used for a diverse range of tasks. It also makes sense to write the system in the same language being analysed.

> C is quirky, flawed, and an enormous success - Dennis Ritchie [4].

1.1 Aim

The aim of this project is to minimise the use of the stack in the hope of producing an implementation of stackless C. This can be achieved via a C source to C source translator which outputs code which does not use the program stack; the heap is used for local variables and control flow data instead. The feasibility and techniques of a solution will be developed and run alongside the investigation of source-to-source systems and appropriate transformations.

1.2 Objectives

It should be considered that many solutions to some of the aforementioned problems already exist in some form. Despite this, there appears to be little discussion of implementing stackless C in a similar manner, or accumulating the solutions and benefits available into one concise project.

1. To investigate methods of implementing stackless C and source to source transformations of the C programming language.
2. To investigate methods of implementing source to source transformations and how they can be applied to this problem.
3. To implement proof of concept source to source transformations which minimise the use of the stack in C. This will consist of:
    - Parsing of valid C source code;
    - Application of transformations to move program data and control flow information off the stack;
    - Output of transformed C source code.
4. To analyse the feasibility and effectiveness of the various techniques considered and implemented.
5. To understand and discuss how this project can be used and how the findings can be applied in related fields.
6. To propose relevant future work which builds on the strengths of this project.

1.3 Document structure

The document is structured as follows:

Chapter 2: Literature Review The next chapter reviews the literature available in this field and other associated, relevant fields. The literature will serve as a part of the background information for this project, outlining the use of stacks, problems associated with stacks, stacklessness, benefits of being stackless and some existing systems.

Chapter 3: Technology Analysis The technology analysis will build on some of the material covered in the literature review. Compiler (specifically parsing) background, techniques and applications are covered, along with other information pertaining to the design and/or implementation of the system.

Chapter 4: Proposed Solution This chapter will set out the requirements of the system and the possible approaches. Discussion of the merits of each method and a decision on the high level design of the system are presented.

Chapter 5: Implementation The implementation covers the actual details of the system - how the transformations are implemented and meet the requirements of the system.

Chapter 6: Discussion An evaluation of the implementation and analysis of the results are provided. The techniques used, problems encountered and successful areas are noted. Future work and methods to develop the system further are presented.

Chapter 7: Conclusion A summary of the project from start to finish, highlighting noteworthy aspects of the system and the overall findings of the project as a whole.

Appendix A: Code Includes results in the form of sample input and output, and some code excerpts from the system.

Chapter 2

Literature Review

Building on the work of others will be a vital aspect of the project, therefore this literature survey will help to uncover what has been achieved by others working both directly on this topic and in related fields. Furthermore, this section of the document should clarify the aims and intentions of the project, as well as providing a greater depth of detail surrounding the field.

This literature review will begin by defining and investigating the ideas and motivation behind *stackless C*, including a discussion on the use and importance of stacks in many modern day imperative programming languages (and the languages which influenced their development).

As previously mentioned, this project is experimental and falls within an under-researched area. The main result of this is that the literature directly relevant to this project is limited and composed primarily from non-academic sources. While there is nothing that directly matches the aim of the project, there is material on similar ideas such as avoiding buffer overflows and co-operative threading which will be investigated to provide ideas and motivation.

The uses of stackless C will then be covered; what new possibilities may arise, what problems can be solved and how this might affect the outcome. Due to the dearth of directly related literature (from formal to informal, published papers to search engine results), it has been necessary to examine how the findings and benefits from work in related fields could impact on this project. Examples include work on application vulnerabilities through to conversions of other languages to be stackless. Existing literature and projects in some of these fields will be evaluated as appropriate.

This chapter starts with some clarification and explanation of the term *stack* and its associated history, followed by an introduction to the notion of *stacklessness*. A section on uses of stackless C and problems associated with the stack further motivates the project, generally concerning security, threading and debugging. Finally, a section is devoted to existing systems which provides some background to the fields of language modification, program

transformation & translation and methods of improving the security of applications.

2.1 Stacks

Key to this entire project is the concept of stacks; what a stack is, its history & *raison d'être*, how it is used and its associated advantages and disadvantages.

2.1.1 What is a stack?

Stack according to the OED[1]:

> In a computer or calculator, a set of registers or storage locations which store data in such a way that the most recently stored item is the first to be retrieved; also, a list of items so stored, a push-down list [5].

A stack is a data structure which represents "a pile, heap or group of things ... with its constituents arranged in an orderly fashion" [5]. More specifically, it stores arbitrary data types and operates according to the *"Last In, First Out" (LIFO)* principle. The core stack operations are *push* and *pop*; the former puts a given element on the top of the stack while the latter removes the top element of the stack. Implementation is often in the form of an array (by means of a contiguous block of memory), with a pointer to the top of the stack. Alternatively, a linked list can also be used.

Types of stack

The term *stack* is widely used and so some context is generally required to identify how a stack is being used, what data it contains, and for what end purpose. When referring to the stack, it will be in the context of how an application manages execution data stored in (a stack of) activation records (see definition in section 2.1.1) rather than the basic abstract data type. These differences shall be briefly covered in the following paragraphs.

Data stack The stack is a ubiquitous abstract data type (ADT) as previously described, used throughout all aspects of a modern computer system from the architecture level (e.g. the call stack), through to a data structure used within an application (e.g. the implementation of a soft-calculator).

Call stack The call stack, also known as execution stack, control stack, program stack (of activation records, containing both control flow and data), function stack, or run-time stack [6] is used by the main modern imperative languages in use today to store activation

[1] Oxford English Dictionary

records as a core aspect of their execution. This is generally implemented as a contiguous block of memory. These activation records track the execution of the current program.

Hardware support is often provided for stacks, in the form of a special (dedicated) register, the *stack pointer*, used to store the memory address of the top of the stack [7]. Some architectures have instructions to directly and indirectly manipulate the stack (i.e. CISC architectures are more likely to have a wider range of stack-based instructions, such as dumping multiple registers to the stack).

On the Intel x86 architecture, there are three main registers used to maintain information about the stack [8]:

EBP (Extended Base Pointer) Stores the address of the current stack frame.

EIP (Extended Instruction Pointer) Stores the address of the current instruction.

ESP (Extended Stack Pointer) Stores the address of the top of the stack.

The stack starts at high memory addresses and grows towards lower memory addresses (as items are pushed onto the stack into the position stored in *ESP*, see figure 2.1), either by CPU specification or ABI implementation. On other architectures, the stack may grow "upwards" [2, 9].

Unless otherwise mentioned, Intel x86 will be the architecture referred to whenever it is appropriate to reference a specific architecture.

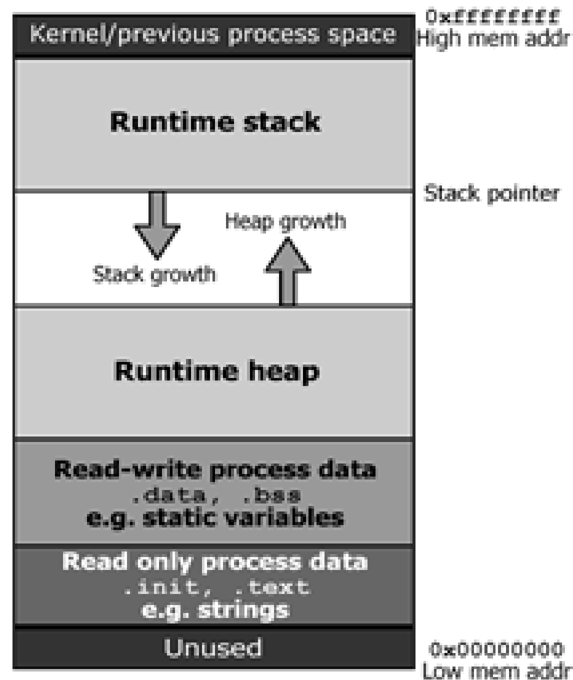

Figure 2.1: Program memory

Protocol stack A protocol stack is an unrelated concept, specifically the implementation of a networking protocol specification or suite. Since this is completely unrelated to this project, this term will not be used henceforth.

Stack computers Stack computers are machines which use memory in the form of stacks and/or a machine which has a zero operand instruction set [7]. Uses include the Java VM and the Forth programming language (see Koopman '89 [7] for more information). These are not relevant to this project and are not discussed further or referred to henceforth.

Other definitions

Heap Similar to the term stack, the term *heap* can refer to different ideas and contexts. The heap data structure, i.e. the tree which satisfies the heap property is unrelated to this project; the term heap is used as in the context of memory which can be exploited by dynamic memory allocation.

The heap is a key entity in this project, since all *auto* variables and control flow data will need to be moved onto the heap (making considerable use of dynamic memory allocation). See section 2.1.3 for further discussion and figure 2.1 for an illustration.

Activation records Subroutine calls require a call stack to store the return address i.e. the location of the next instruction to execute once the called subroutine has finished executing. A location to store data when passing parameters to a subroutine and storing local variables (when recursion is permitted) is also needed. A solution is the use of *activation records*, which are generally used to pass data to a subroutine being called. The required data is generally stored on a single stack within an activation record and typically consists of local variables, actual parameters and a return address [1]. Two types of data are stored in an activation record - the control flow data and the program data. These data types are distinct and ideally would be kept separate since hardware cannot tell the difference between control flow and subroutine variables, leading to problems covered later (section 2.3.1). Rather than have two stacks, one for each type of data, it is combined and stored in an activation record on a single stack for issues of locality. A *stack frame* is a block of data on the stack and is sometimes used interchangeably with *activation record*.

2.1.2 Why are stacks used?

The program call stack is used to store a list of activation records which have a machine-dependant structure. Data stored in CPU registers can be accessed faster than data stored in any other memory location (cache, RAM, optical & magnetic devices) [10]. Only a very small subset of all programs will be able to store all required program data in the registers available; modern desktop architectures typically have in the order of 16-32

numeric registers [11]. The stack allows program data to be stored in memory in a non-static manner. This resolves the issue of having an insufficient quantity of registers to store all program data, as well as allowing a flexible method of subroutine execution (which permits recursion).

The use of a program call stack is however, not without its associated problems. Stack overflow will occur if too much data is stored on the stack, exceeding the stack's size which was allocated when the application loaded. Uncontrolled recursion (which is not tail-call optimised[2]) is often the cause of this problem.

In computer systems and programming languages which have insufficient protection against memory safety violations stack buffer overflows can occur, causing at best data corruption and at worst program termination or arbitrary code execution. Such security implications and methods of preventing this are discussed later in section 2.3.1.

History

The idea of using stacks in programming has been around for some time; stacks can be used directly to improve execution efficiency, and indirectly allow a number of enhancements to programming languages. Since the late 1950's, stacks have had a degree of hardware support, as shown in table 2.1.

Originally, stacks were not present in computer programming languages, however their use allows some of the shortcomings of languages such as FORTRAN to be overcome, such as the inability to recurse (since all subroutine variables were statically allocated).

Language	First Appearance	Recursion Support
FORTRAN	1957	No
ALGOL	1958	Yes [13, 14]
COBOL	1959	No
PL/I	1964	Yes
BCPL	1966	Yes
B	1969	Yes
C	1972	Yes

Table 2.1: Recursion support in earliest versions of programming languages

2.1.3 How are stacks used?

The use of activation records stored on the stack permits variables which were once statically allocated to be automatically allocated within this block of data, such that the lifetime of

[2]Tail call optimisation allows recursion with the benefit of being able to discard the current activation record provided the last call in the function is to itself. This allows recursive functions to act as iterative loops and prevents the stack from overflowing [12].

these variables is equal to the lifetime of the subroutine execution. Whenever a subroutine is called, a new activation record is pushed onto the stack (and popped off when the subroutine returns) - this activation record has a "new" set of automatic variables if this is a recursive subroutine call.

Unless recursion or an advanced subroutine calling mechanism (e.g. co-routines) are required, a stack may not be required for program execution because static variable allocation will suffice. These characteristics have a more complicated control flow, which in turn requires the control flow data to be stored somewhere - within the activation record.

The main benefits associated with the use of stacks are directly related to the use of program memory:

Storing the subroutine return address Whenever a subroutine is called, a return address is required such that when the subroutine has finished, control can be handed back to the calling function which can continue executing instructions from the return address and therefore allows recursion. This is the main reason to use a stack.

Passing subroutine parameters When a subroutine with formal parameters is called, it will require actual parameters (i.e. real values) to be specified. These values are stored at the beginning of the new activation record generated for the called subroutine and can be used within this subroutine until it returns, at which point the activation record will be popped from the call stack.

Dynamically allocated local variable storage for subroutines Variables local to a subroutine are automatically allocated, meaning that space is allocated for all local variable storage on the stack (within the activation record) whenever the subroutine is called. This data is unique to this particular subroutine call so whenever the subroutine is called again, it will have a "fresh" set of variables. Storing local variables in this manner rather than statically allocating memory permits recursion.

Memory management In C, memory can be managed in three different ways. Statically allocated variables last for the entire duration of the program execution. Automatically allocated variables (signified by the use of the implicit keyword *auto*) are stored on the call stack in activation records and exist until the stack frame in which they are allocated is popped off the stack. Dynamically allocated variables are stored on the heap and allow for programmer (or program) controlled lifetimes, by manual allocation and deallocation.

Blocks of memory can be allocated on the heap via a call to *malloc* (or *calloc*) and released with a call to *free*. Alternatively, a garbage collector such as Boehm-Demers-Weiser [15] can be used to perform automatic garbage reclamation. New items on the heap have higher memory addresses than older items; the heap grows upwards, unlike the stack [16].

An advantage of stack based memory allocation is its relative speed over heap based memory allocation. Since an underlying aim of the project is to move data off the stack and onto the heap, a performance hit is to be expected. The extra complexity associated with

the memory management of putting everything onto the heap can hopefully be mitigated through the use of garbage collection, although this will often have some organisational overhead associated with the memory deallocation [17].

It has been claimed by Appel [18] that provided there is sufficient memory on a machine (such that a stop-and-copy algorithm can be sensibly used), then garbage collection of the heap is a cheaper operation than popping an element from the stack. Miller & Rozas [19] subsequently disagreed with Appel's findings - while the premise that garbage collection can effectively be "free", it was not the case when used for continuation frames on any tested architecture. The stack has the benefit of its memory being immediately available for reuse since live data is instantly removed, as well as having fast stack based instructions on a number of architectures.

2.2 Stacklessness

This section is dedicated to discussing the concept of being *stackless* in more detail.

2.2.1 What is it to be stackless?

Part of this project will aim to alter the form of the control flow data of the program. It should be noted that there is a difference between separating the control stack from the data stack (within the context of the entire program stack) - in machine architectures such as the x86, there is only one program stack which contains activation records which store both variables (program data) and memory locations (return address and control flow).

Dynamically allocating all local variables on the heap rather than being automatically stack allocated, will result in the stack containing just function parameters and control flow data. If the control flow data can be moved to the heap in the form of dynamically allocated activation records, then the use of the program stack can be minimised or removed almost entirely.

Stackless C will still have the power of "traditional" C: recursion will still be supported. The stack will still essentially exist, whether it is hardware or software managed. Nowadays, all stack manipulation is hardware managed through the use of push and pop instructions, whereas previously it was all software managed and hidden by the compiler. Once again, the stack will be software managed to produce stackless C.

In *Stackless Python* (a Python interpreter which makes calls written in C), the C stack is not used. Stack frames are manipulated "manually" instead. This is comparable to the aim of the system to be implemented. Stackless Python is discussed in more detail in section 2.4.2.

2.2.2 Why stackless?

Becoming stackless can have a number of benefits. Even if these benefits are to be ignored, then this project will consider the techniques which can be used to achieve this aim and effectively combine the ideas from a number of existing systems, all with distinct aims, into one system.

Some of the benefits shall be discussed further in the next section (section 2.3):

Security Program stacks are susceptible to a number of exploits. Being stackless could help mitigate this problem.

Miscellaneous The ability to debug a multithreaded program more easily, which can currently be problematic. If the program's data (control flow and variables) are stored on the heap, they can be more easily inspected and modified. This is because there is software management of the call and data stack which allows software control of the entire program, i.e. a program can be programmatically modified. C's scoping rules can be modified, and continuations & co-operative threading can be implemented.

2.2.3 Techniques of implementing the stackless language

The primary software deliverable of this project is an application which delivers stackless C in some form. This can be achieved by one of three main methods:

Compiler backend modifications The backend of a compiler could be modified such that the assembly code generated minimised the use of the call stack, by allocating data on the heap. This technique would probably be combined with initial modifications to the code at the abstract syntax tree (AST) level. It is expected that this would provide a thorough and effective solution, however the modifications would not output C source and the learning curve/difficulty of the project is likely to be high. A compiler to be modified must be sourced - a full-blown commercial grade compiler such as GCC[3] is unlikely to be a good choice due to the complexity involved in modification. It would also be necessary to recompile all code with this modified compiler.

Text replacement A set of text replacement macros, both simple and sophisticated, could be used to move local variables onto the heap. It seems likely that this approach may struggle at the control-flow modification step, resulting in increased complexity & decreased clarity at best, and attempting an impossible task at worst. The use of a macro language such as m4[4] could be used for this translation technique.

[3]GNU Compiler Collection, `http://gcc.gnu.org/`

[4]Macro processor, `http://www.gnu.org/software/m4/m4.html`

AST transformation Leveraging existing tools such as Flex[5] and Bison[6], an application could be written to operate on the abstract syntax tree constructed with the help of these tools. Translating the C source to an AST (via the steps of lexical analysis, parsing and semantic analysis [1, 20]) which undergoes a translation to produce a new AST, and is subsequently converted ("compiled") back to C source (which can be fed into the C compiler of choice). This is the technique of choice for this project.

2.3 Uses

2.3.1 Security

Security problems arise through buffer overrun attacks (such as smashing the stack [2]), which exploit the combination of control flow and program data in the same structure (i.e. the activation record). Stack overflow can occur (particularly on Windows systems) when the stack data exceeds the memory available for the program stack which requires a block of contiguous memory since the stack is comparable to an array.

A stackless C would circumvent a number of problems with memory limits and enables various methods to defend against buffer overrun attacks to be implemented. The result would be a more "secure" executable - a useful and desirable result, although at the expense of speed. The notion of being stackless can refer to both the data stack and call stack, i.e. local variables and control flow data on the stack. Ideally, both should be moved from the stack.

There are no perfect techniques for avoiding buffer overflows in C, and it is likely that no perfect solution exists. Current solutions can come at a high performance cost, are incompatible with legacy code (e.g. Cyclone) or require either hardware or software modification (e.g. recompilation based tools), either of which may not be possible or desirable depending on the program use and environment. Essentially, safe execution depends on some combination of language design, compile time and runtime checks [21].

A number of solutions do exist which provide some extra safety and/or security at little cost but the problem is fundamental to the design of C. Unless this problem is rectified in C which will come at some inevitable cost, these solutions will merely be patching the symptoms rather than the problem(s). The existing approaches are rarely complete or bulletproof, yet they still increase the difficulty in executing a successful exploit, or reducing the available attack vectors.

---

[5] Fast LEX (lexical analyzer generator), `http://flex.sourceforge.net/`

[6] GNU version of YACC (Yet Another Compiler Compiler), `http://www.gnu.org/software/bison/`

Buffer Overflow Attacks

Buffer overflow attacks are a memory safety violation and a vulnerability which can be found in programs implemented in languages that do not provide bound checks for variables. Within runtime environments for languages such as Java, C#, Pascal and Ada do not expose themselves to buffer overruns, since variables are bounded and checked appropriately [22]; C and C++, however, are susceptible. Buffer overflows have been the main form of security vulnerability for the last ten years because they are both relatively common and easy to exploit [23].

Two goals must be met to achieve a malicious (and non-DoS[7]) exploit: a change in the control flow of the program and the injection of some code which is subsequently executed. Attacks exist on both the stack and the heap, e.g. *smashing the stack* can also be modified to result in *smashing the heap*.

As has already been established, the translation which must be applied to the input source code can be crudely reduced to moving all local variables and control flow data onto the heap. Consider just moving the local variables to the heap - this would sidestep some of numerous buffer overflow attack vectors, specifically *smashing the stack*.

Any buffer in C can be overflowed i.e. data can be written in *and then past* the memory allocated for the buffer due to the absence of array bound checks.

Baker & Pincus [22] provide a clear and concise discussion covering the main buffer overrun exploits which are described here.

Smashing the stack Originally described by computer specialists Elias Levy and Christien Rioux, more often referred to by their online handles of AlephOne and DilDog, this technique takes advantage of the proximity of local variables and return addresses for a subroutine stored in an activation record on the same stack [2].

As previously mentioned, the stack contains activation records whose structure is such that a buffer overflow in the local variables will overwrite other local variables and then control data stored in the same activation record. Smashing the stack entails corrupting the return address which is stored on the stack within the current activation record, just after the local variables. When the subroutine comes to return, control is returned to the corrupted return address which now points to a location in memory of the attackers choice, typically to some injected arbitrary malicious code which is subsequently executed [2, 23].

The two steps of code injection and address corruption can be performed in one overflow using a specially crafted array (provided the buffer is of sufficient size); alternatively, the payload and address corruption can be delivered at different times within the lifetime of the program.

[7] Denial of Service

Return-to-libc (arc injection) Existing code can be used for nefarious purposes too. If control can be transferred to code which already exists in the application (i.e. application code or libraries used), a new arc can be inserted into the control flow graph (c.f. a new node is inserted into the control flow graph in the case of stack smashing). Typically, the return address of a subroutine call is changed to point to some existing code e.g. *exec()* within *libc* and the location of some attacker provided code is passed as an argument to the called function (parametrisation). This form of attack can bypass non-executable stack protection mechanisms.

Pointer subterfuge A function pointer can be stored statically, dynamically (on the heap) or automatically (on the stack). If there is a buffer which is adjacent to the function pointer in memory, then the function pointer is exposed to the risk of being corrupted. It is possible for the function pointer to be changed such that attacker supplied code which will be executed when the function pointer is called.

Other variations include data pointer corruption, provided a memory address is known or exception handler hijacking in which the function pointer called when an exception is raised is changed to an arbitrary memory location, allowing control to be transferred elsewhere [23]. VPTR smashing only affects C++ and therefore shall not be discussed.

Other attacks such as smashing the heap and longjmp buffer corruption also exist.

Security, vulnerability detection & exploit prevention

Buffer overflows are possible because of the mixing of program data and control flow information. This means that program control information can be treated as (writable) data, allowing it to be modified provided a number of conditions are met. A combination of the available solutions can provide an extremely effective defence against the exploit vectors currently in use; a system/program which just uses one solution is likely to still be susceptible. Cowan et al [23] found that Stackguard and an NX-flagged stack (see below) provide "substantial coverage of the field of possible attacks". They also discuss a selection of the defences available against buffer overflows.

NX bit The NX (No eXecute) bit, also known as the XD bit, is a hardware solution for preventing a number of buffer overflow attacks by introducing executable space protection. The concept of executable space protection makes execution of instructions and storage of data mutually exclusive operations within any location in memory. A memory region with this bit set is marked as being available for storage i.e. it can be freely written to, and any attempt to execute instructions from such an area will raise an exception. Unfortunately, not all program data segments can be marked as "NX" because program compatibility can be significantly affected.

Software implementations which emulate the NX bit are present in most modern operating systems under various names, including WˆX (OpenBSD), PaX (Linux) and Data

Execution Prevention (Windows).

There are other routes available to protect against, detect or render non-trivial a number of exploits and vulnerabilities in software in use today.

Address Space Layout Randomisation (ASLR) ASLR changes the locations of core elements of a program - the stack, instructions, heap and libraries are stored in random locations within the processes memory. The purpose of the random locations is to make locating these elements harder, therefore it is harder for an attacker to find the memory address of any (shell)code injected or the location of libc for an arc injection attack. It should be noted that it is still possible to locate the desired data because this technique forms just part of a defence in depth strategy rather than being a complete solution.

ASLR is used as part of a wider security system. There are implementations for *nix operating systems, as well as the latest versions of Windows and Mac OS X.

Compiler checks Two forms of compiler checks can be performed: bound checks on all read/write operations on arrays. This will ensure that corrupt or invalid data cannot be read from the array, nor can the array be overflowed to corrupt adjacent data. This is commonly found in type-safe languages, Purify and patches for various compilers exist to implement this form of check [23]. The other check that can be performed involves detecting corruption (and preventing execution). Checking the integrity of pointers prior to dereferencing and inspecting the stack can both be used to achieve this indirect form of check.

Dynamic detectors Despite the prevalence of buffer overflow exploits, dynamic buffer overflow detectors are not widely used because of their inconvenience and shortcomings. Such detectors cannot detect or protect against all form of attacks, yet may come at a high performance cost and even be incompatible with existing code [24].

Systems such as StackGuard [25] and ProPolice [26] can detect and prevent stack smashing attacks through the use of "canaries". Generally, the layout of the data within the stack frame is modified to include "canary words" at appropriate locations e.g. after stack allocated arrays and before control flow data. A buffer overflow will corrupt a canary; when a canary is checked and does not validate against the expected value then an exception can be raised. Only the stack is protected against stack smashes; data on the heap is still vulnerable to heap smashes.

StackGhost [27] is a kernel modification for SPARC systems which protects the return address on the stack. It is implemented through a XOR cookie (which detects modifications made), an encrypted stack frame and a separate return address stack. While there are problems with this technique, the overhead is small since it is a kernel modification, resulting in an effective solution at a small cost which can be used to reduce the likelihood of a successful exploit.

Runtime dynamic libraries Libsafe and libverify are libraries that provide both static and dynamic protection. Library functions which contain potential problems are statically patched and runtime range checks are added to ensure that control flow data is not overwritten. Libverify is an enhancement of libsafe which also implements a number of other security features such as the use of canaries. This requires each function to be copied to the heap where it can be modified and accessed through wrapper functions (although this step is only required on Intel architectures) [16].

Hardware modifications SmashGuard [28] is an example of a system which aims to provide protection through hardware modification. Such systems have a separate hardware return address stack and may modify how procedures are called and return instructions used [29]. This provides dynamic protection against return address corruption without any code overhead or programmer effort. Further advantages include a minimal performance impact, compatibility with existing code and no requirement to recompile.

Others Systems of note which have not been mentioned and will not be discussed include CRED (C Range Error Detector), RAD (Return Address Defender), PointGuard, FormatGuard & StackShield. These are mentioned for completeness.

2.3.2 Threading & Debugging

Gustafsson describes the current situation regarding threading in an article titled "Threads Without The Pain" [30]. Operating systems are multithreaded today, allowing multiple flows of control to exist at any given time (which may or may not result in actual concurrency). The majority of OSs use pre-emptive scheduling to determine the order and period for which a thread can execute. In this method, the scheduler starts and stops threads once they have executed for some fixed period of time thereby preventing any single thread from monopolising the CPU, although this comes at the expense of a running thread being stopped at any moment. Generally, especially in a desktop environment, this method of scheduling works fine despite its associated problems. There is, however, an alternative style: co-operative threading (also known as non-preemptive, synchronous threading, co-routine based programming). Co-operative threading entails a thread relinquishing the CPU after some time; this voluntary yielding will typically be done at defined points, such as when the thread requires another thread to execute or it is an appropriate point to pause. Many problems typically associated with multithreading (which in fact are problems that exist due to the "defect" in pre-emptive threading) are bypassed - locking is generally not required and race conditions can be easily avoided.

Another issue which arises in multithreaded systems is that of a fixed size stack created for each thread and the associated tradeoff required to balance between "wasting" memory[8]

[8] In the sense that a potentially significant amount of memory has been allocated and may never be used during the lifetime of the thread

and risking stack overflows due to insufficient stack space. This problem is most notable in applications with a significant number of threads, or systems with a small amount of memory.

Unfortunately stacks are not one of the many data structures which now have a dynamically sized equivalent. A solution to using fixed size stacks is to convert the language to have a stackless variant in which all information previously stored on the stack is stored on the heap instead. The programmer can then make the choice between threads which don't have a fixed stack size (resulting in lightweight threads) at the expense of a slight performance hit. Currently, "fully integrated support for co-operative threads - not to mention support for mixing preemptive and non-preemptive threads - is rare" [30] and modifications such as Stackless Python and Stackless C provide a form of alternative for the time being.

Storing the control flow data and local variables on the heap will allow for easier inspection of the state of the program and may benefit the debugging of multithreaded programs.

2.3.3 Development of transformation and source-to-source systems

There may be a range of other benefits from this project, either through the findings or the output of the application which are beyond those considered here. Any notable findings shall be included in the discussion (chapter 6).

2.4 Existing systems

It should be stated that there appears to be no directly comparable project or existing system. This project will likely result in a combination of ideas from related projects, such as the CIL[9] C source translator from Berkeley and Protothread's[10] stackless threads.

A number of modifications of C exist. There exist safe dialects of C such as Cyclone[11] from AT&T Research which extends the language as well as helping to guard against some of the problems associated with C (e.g. dangling pointers) [3]. CCured is a C source to C source translator which inserts memory checks, built ontop of CIL.

2.4.1 Type safety

Type safety is a key concept required in a programming language to ensure safe execution of a program. If type safety is present, a program will execute faithfully according to the language semantics *and* be prevented from corrupting itself in some way through the use of both compile-time and runtime checks.

[9] C Intermediate Language, `http://manju.cs.berkeley.edu/cil/`

[10] `http://www.sics.se/~adam/pt/`

[11] `http://cyclone.thelanguage.org/`

C is the archetypal type unsafe language and its use of raw pointers allows a program to overwrite its own stack and generate memory exceptions (e.g. segmentation faults). Preventing such problems and ensuring type safety can be achieved through different techniques [21]: language design, compile time checks and runtime checks.

CCured & CIL

CCured [31, 32, 33] is a source-to-source translation system designed to add safety to C by adding the minimum number of runtime checks to prevent all memory safety violations. The project is based on the *CIL (C Intermediate Language)* framework (also from the same group), itself implemented in OCaml[12].

The CIL framework can be used for any form of source-to-source translation or analysis. It is undoubtedly a powerful translation tool capable of not only clarifying ambiguity and removing repetition & redundancy, but maintaining a close relationship with the original source code and simplifying the semantics of the source program. These attributes mean that the input source code is translated to a highly structured subset of valid C code with the aforementioned properties allowing it to be more easily manipulated. The operations performed in converting C to CIL include converting GCC extension code to regular C, scoping of variables correctly (by renaming and relocating their declarations), ensuring function declarations are present, constant folding, normalisation of type specifiers and a range of other transformations - see the manual [34] for more information. All source C is broken down into control flow data, side effect data (variable assignment + function calls) and other expressions (which are neither control flow, nor have side effects). A suite of modules are provided in the form of a library; amongst these are the directly relevant modules which implement StackGuard (as per definition [25]) and "heapify" a program, as well as liveness analysis and even converting to C++ code.

On top of CIL, CCured analyses and monitors pointers carefully to prevent operations which could lead to runtime problems. The type of each pointer can be inferred by its usage within the source code. The following pointer categories exist:

Safe Pointers which are only dereferenced. No casts or arithmetic is permitted. These are the safest and preferred type.

Sequence Pointers which allow arithmetic (and can further be categorised into pointers which only move forwards, such as those used when progressing forwards through a block of memory and pointers which move in both directions). These are bounded to ensure their safety.

Wild Any other pointers are classed as wild pointers, and due to their increased capabilities they incur a higher cost due to the checks required to ensure memory safety violations are not committed.

[12] An object-oriented implementation of the Caml dialect of ML, `http://caml.inria.fr/`

As ever, CCured comes with a number of trade-offs. The application is only semi-automated - it is necessary to guide the application by identifying pointer types in some cases, where such information cannot be gleaned from the program source code analysis. The runtime performance loss is somewhere in the region of 10-60%, which when compared to some other systems can be considered reasonable.

Cyclone Cyclone [3, 35] is a safe dialect of C, with the aim of having comparable safety to that of Java whilst retaining the syntax, types, semantics and idioms which C programmers are accustomed to. As such, all changes made are for the purpose of safety - these consist of static checks and the addition of dynamic checks into the code as required. Since some functionality is removed due to the restrictions imposed on C, a number of features are added to regain that which was lost. In many aspects, Cyclone can be compared to CCured - CCured makes use of a garbage collector, removes some control from the programmer in terms of data representation and has varying operator costs depending on the runtime checks inserted. The main difference lies in the differing philosophies - Cyclone tries to "preserve C's hallmark control", whereas CCured has a reasonable porting system which adds extra fields to data representations and more runtime checks over checks via static analysis.

2.4.2 Concurrent programming

Peschanski & Hym [36] mention some of the pros and cons of stackless architectures for concurrent programming in their paper on a runtime environment for a pi-calculus. Processes are simpler and more lightweight without a stack - it would be possible to calculate an upper bound on the memory required by a process at compile time and resource management within a virtual machine is simplified. The simulation of stack frames are sufficient to handle recursive functions in the absence of the stack and have the added advantage of having no stack limit.

Stackless Python

Stackless Python[13] is an implementation of Python which is decoupled from the C stack to enable a different paradigm. Not using the stack in Python can reduce the memory footprint of the application and add improved co-routines & generators to the language, both of which are based on continuations. A machine independent implementation of continuations requires a stackless implementation because the Python interpreter relies on the execution of some C code for all Python calls (which uses the C stack). The paper [37] outlines a number of reasons for removing the C stack, including removing the limit on recursion depth, limiting the possibility of out-of-order execution, the ease at which the C stack can be decoupled and the speed gained from using co-routines. Stackless Python

[13] `http://www.stackless.com/`

claims to “allow programmers to reap the benefits of thread-based programming without the performance and complexity problems associated with conventional threads” through the use of lightweight threads. This project is a modification to Python rather than a modification of C. The Python interpreter relies on C function calls and is written in C, so this project changes how Python uses the underlying C.

Protothreads

Protothreads [38] are lightweight threads implemented within C designed to improve concurrent programming for embedded systems i.e. systems which are memory constrained. For this reason, protothreads are stackless and all run in one system thread. As lightweight threads, protothreads draw upon concepts from threads and events - blocking & waiting, and low memory consumption & stacklessness. Implementation is primarily through the C preprocessor and can make use of the GCC C extensions.

2.4.3 Translation & rewriting systems

Marst

Marst is a GNU tool which provides source-to-source translation from Algol 60 to C. Documentation and references in any form to this tool are few and far between - the results from searching seem to indicate that this is not a widely used tool.

The documentation associated with the tool states that the translator uses a recursive descent technique for parsing, with the translation performed in two passes. Block structure and identifier information is gathered on the first pass; output C source code is generated directly from the source on the second pass. The result is C source code which defines the same language [39]. Unfortunately, no more detailed information is provided about the technique used to do the translation, nor an analysis on the performance or results of using Marst.

Binary rewriting

Prasad & Chiueh describe a binary rewriting approach against stack based overflow attacks for the PE executable format [40]. Such an approach bypasses the requirements of having the source code, which is required for a large number of techniques used to prevent buffer overflows. There are a number of cases when having the source code is not feasible, such as legacy software and software from some vendors. Static binary rewriting can be used to add a return address defence (RAD) mechanism to the application being modified and ensuring the integrity of the return address. The main problems which arise when using this technique are where the instructions should be inserted (since the disassembled code will not be as structured as the original high level source code) and how to minimise any side effects of inserting code (on memory addresses already specified within the assembly

code). The operation of this system is via a recursive traversal algorithm for control flow analysis to establish where code must be inserted.

Larus & Schnarr [41] note that all binary rewriting systems are for the purpose of "emulation, observation, and optimisation". Their paper discusses another binary rewriting system known as EEL for RISC architectures - this is not covered in further detail here since the previously described system implements a security mechanism, whereas this system is a more general purpose binary editor.

Other existing binary rewriting systems included for completeness are ATOM [42], Etch [43], LEEL [44] and UQBT [45].

2.4.4 Tools & frameworks

A list of tools and frameworks which exist to provide functionality related to parsing, tree building, representing & manipulation and similar appropriate areas is provided, with brief notes and sources of information for each. This list exists because the tools & frameworks are of direct relevance to the project, or simply for completeness and consideration for future or related work.

SableCC [46], JavaCC [47] and ANTLR (ANother Tool for Language Recognition) [48] Parser generators written in Java, which claim to have some support for C. SableCC is a powerful parsing system, but support for C is provided by a third party for an outdated (and unavailable) version.

ANTLR is a well developed system, originally created by Terence Parr. The latest version of ANTLR (version 3.01 at the time of writing) is an overhaul of a successful and powerful set of tools for generating parsers. There are grammars for a number of languages and code can be generated for runtime execution in C, C++, Java, Python, C# & Objective-C. A basic C grammar is provided and much of the complexity of constructing parse trees is removed. Unfortunately, writing code to perform transformations on these trees was difficult and much of the documentation is provided in a book which must be purchased.

CTool & ckit [14][15] ckit is written in SML and produces an AST of SML datatypes from C source code. It is no longer under development and has "rudimentary" documentation. This is not used because it is both inactive and does not use an accessible programming language. CTool includes a lexer and a parser in the form of a library to do source-transformations. It is under active development, but is not mature enough to support this project. It has potential for similar systems or future development.

[14] http://ctool.sourceforge.net/
[15] http://www.smlnj.org/doc/ckit/index.html

Elkhound & Elsa [49] Elkhound is a parser generator and generates parsers using the GLR algorithm with any context free grammar, thereby enabling unbounded lookahead and providing support for ambiguous grammars. Primarily for C++, but appears to be a useful tool.

Safe C Compiler (SCC) [50] An optimising C-source to C-source compiler which implements the appropriate checks to detect memory access violations.

LTC (Linguistic Tree Constructor) A system which just builds linguistic syntax trees from text, either top-down or bottom-up. While related to trees, this is not particularly appropriate for this project.

Harmonia A framework for language-aware programming tools. Not particularly relevant to this project

GOLD Parsing System [51] Windows based tool which provides both lexer and parser generating capabilities for a large number of languages. This is made possible through the use of intermediate "compiled grammar tables", generated from grammar files in a BNF-based format, and read by an "engine" implemented in the target language. Lexical analysis uses a DFA and parsing uses an LALR algorithm.

SUIF (Stanford University Intermediate Format) [52] A compiler framework from Stanford written to aid research projects. An inactive project with documentation, albeit slightly inaccessible. This system would require considerable learning and is focused towards compiler optimisation and research based compiler implementation.

FLICK (Flexible IDL (Interface Definition Language) Compiler Kit) [53] Contains CAST (C Abstract Syntax Tree) representation which provides an internal representation of C or C++ source code. Since it is used for IDLs through RPC & RMI, its focus is not particularly appropriate for this project.

Lemon [54] An LALR(1) parser generator for both C and C++, but uses a different grammar syntax, with the aim of reducing coding errors. On top of this, the parsing engine claims to be faster, thread-safe and re-entrant - features which YACC & Bison cannot claim. Further differences include the ability to simultaneously run multiple parsers and an inversion of the calling mechanism - the tokeniser calls the parser. Lemon was written for and maintained as part of SQLite, who share an author. It is both written in, and generates a parser in C.

Ragel [16] State machine compiler and lexical analyser builder. Can generate code for C, C++, Objective C, D, Java & Ruby.

Coco/R [55] Compiler generator. Coco/R generates a lexer and parser for the language specified by a grammar. Supported programming languages are C#, C++, Java + others.

TreeDL [17] Language to describe tree data structures & associated operations. Helps to eliminate inconsistencies and errors between trees at different stages of the compilation process. Independent of programming language because it documents an interface between subsystems.

The majority of these tools were discounted because they are not sufficiently mature, are simply not directly appropriate to the aims of the project, or would require a significant investment of time to learn therefore incurring a large overhead. The chosen tools are discussed in the technology analysis (chapter 3).

2.5 Summary

This literature review has shown the reasons and benefits of using stacks to store both control flow and local variables, followed by the motivation for becoming stackless - the purpose of the project. There are a number of problems which plague applications written in the widely used C programming language. Many of these exist due to the lack of type safety and bound checking within the language, allowing buffer overflow attacks and other forms of data corruption within a program. The mixing of control information and program data in a structure which is both executable and writable (i.e. the stack) is a key underlying concept for many of the aforementioned problems.

The problems are discussed in further detail and it is shown that being stackless can prevent some problems and enable a number of benefits and new approaches to writing programs. Development and testing of ideas which can be applied to programs are core objectives of the project. Existing systems which combat security issues in programs in a variety of ways are investigated along with other relevant systems, including stackless implementations of other languages, rewriting and transformation engines. The tools and frameworks on which this project can build are investigated to determine which are the most appropriate to leverage in this project.

By considering the techniques used in existing systems which modify source code in a similar manner, as well as the background research carried out, a C-source to C-source translation tool could be successfully implemented (as per section 2.2.3). This tool would deliver a number of the specified benefits (section 2.2.2) and achieve the main objectives of the project (section 1.2).

[16] `http://www.cs.queensu.ca/~thurston/ragel/`
[17] `http://treedl.org/`

Chapter 3

Technology Analysis

Before discussing the aim and methods of producing stackless C in any considerable detail, some background on the required technologies and high-level approaches to this project will be provided. Some compiler-related background will be covered, explaining some of the general concepts and noting any project specific issues. The tools available, the types of program transformation which exist and some of the more relevant and/or important elements of the C specification will appear before the techniques which can be used to produce stackless C are examined.

The problem of solving stackless C is non-trivial because the transformations required restrict the code to a subset of the language. Simple text replacement or regular expressions cannot be used to transform the source code. Instead, it is necessary to parse the code and analyse the resultant tree, with the aim of identifying which aspects of the code must be modified. Due to the design (and power) of C as a programming language, many of the cases require the modification, insertion or deletion of statements and expressions within the parse tree. Tree manipulation therefore becomes an important aspect of the transformations. Key to this project is the successful combination of parsing, tree manipulation, transformations and code generation.

3.1 Compiler frontend

The program will essentially include a number of the steps of a compiler, particularly the frontend which is responsible for analysis, coupled with synthesis and output of stackless source code. Some brief background is provided for the relevant stages. A brief description of the relevant stages of the compiler are covered and any notable implementation details.

3.1.1 Lexical analysis

Lexical analysis is used to match characters from the input stream so that they can be grouped together and recognised as a token/lexeme. This process produces a sequence of tokens from a sequence of characters. Patterns of text to match are usually represented by a regular expression. Lexemes are recognised within the input text and tokens for the lexemes matched are produced.

It is typical to associate an integer value to a token type. In the case of keywords in a programming language, a simple token is sufficient, which can be identified by an integer identifier for this token type. For other inputs, such as strings, the text which matched the rule, i.e. the lexeme, must be associated with the token produced. This is necessary because it is important to recognise that the token is a string, but also to be able to access the lexeme.

A scanner is used to recognise the set of characters which match a rule. This is often implemented as a DFA (Deterministic Finite Automaton) which recognises as many characters as possible for a given rule (i.e. a greedy algorithm by default).

3.1.2 Syntactic analysis

Syntactical analysis, also referred to as parsing, is the act of matching a sentence against a grammar to deduce the structure and content of the sentence. The structure of the sentence is dictated by the grammar and may include or be guided by any syntax present. The content of the sentence is any non-syntax parts of the sentence which can subsequently be recognised as having a role or meaning within the sentence during a semantic analysis stage. This later stage deduces the meaning of the sentence which has been established as being valid in the language.

More specifically within a compiler, parsing refers to the matching of blocks of input source code against the rules to establish that the input is correct (i.e. that it matches the grammar which specifies valid sentences within the language being parsed). Some form of hierarchical structure is often used to store the sentences as they are encountered, typically producing a parse tree, or (concrete) syntax tree.

3.1.3 Semantic analysis

Semantic analysis refers to the interpretation of parsed data and generation of information associated with it. This ensures that the input has a valid meaning, whereas syntactic analysis verifies that the format is correct. The responsibilities of this stage include building environments, performing type checking and annotating the parse tree with information to aid the middle and back-end of the compiler.

This system has no need to perform type checking because the input is already assumed to be valid C source code. The primary task of this phase is annotating the tree with infor-

mation about environments, function definitions and variable usage which is subsequently used to transform the tree.

3.1.4 Transformation

Using the annotations generated in the previous stage, the transformations can be applied to the tree to achieve the aim of minimising use of the stack. The transformations operate in-place on the abstract parse tree; any definitions that must be maintained are stored in annotations to the tree. The transformations generate new nodes, replace and remove existing nodes and provide further annotations. See section 4.5 for further information.

3.1.5 Code generation

This stage of the transformation produces an external version of the internal representation of the source code - the tree manipulated by the transformations is transformed back into C source code. Within a traditional compiler, this stage would form a late stage of the compiler backend, performed after code optimisation and conversation to an intermediate form, such as TAC (Three Address Code). Since there is no intermediate representation nor optimisation stage present in this system and the system is to output C source code, this stage is somewhat different to code generation to produce a binary.

For all of the constructs within C (after sufficient semantic analysis), a set of mutually recursive functions can be used to output the C source code from the internal tree representation, without consideration or lookahead to the contents of any subnodes. Each node can generally be "blindly" transformed to code, by transforming both children and adding the appropriate keyword(s) and syntax - essentially walking the tree from left to right. However, in a few cases, it is necessary to check for the existence of either or both child nodes, since these nodes may or may not require certain syntax. This latter type of node must be considered as a composite of not only its type, but also the existence of its child nodes. In reality, this poses no problem since an additional conditional is required.

3.2 Parsing

While parsing is an important aspect of the implementation of this system, this section does not consider it in great detail. Any relevant details regarding the implementation appear in subsequent chapters - only some relevant background is provided here. It should be noted that there is little which is particularly innovative or exceptionally noteworthy with regards to parsing. The subject is well studied and therefore this aspect of the application is predominantly an application of the current technology.

3.2.1 Grammars

A grammar is a description of which set of strings (which constitute sentences) are valid over an alphabet; the set of valid strings (which consist of a sequence of characters from the alphabet) is known as the language for which this grammar is defined. Formally, a grammar is defined as a set of terminal and non-terminal symbols, a start symbol and a set of production rules which provide a transformation from at least one non-terminal symbol to another set of symbols. A valid sentence can be deemed valid if it can be constructed from the start symbol and some applications of the production rules, resulting in a sequence of non-terminal symbols matching the sentence being tested.

Context free grammar

Of interest in the field of programming, is a particular type of grammar - the context free grammar (CFG), developed/identified[1] by Noam Chomsky in the mid 1950s. In a CFG, every production rule consists of exactly one non-terminal symbol being transformed to zero or more terminal or non-terminal symbols. The restriction makes the grammars simpler since they consist of blocks and can be built recursively. This is advantageous because the production rules can be followed without consideration to the current context in which the tokens lie and considerably more efficient algorithms are permitted.

Regular grammar

Regular grammars are a subset of context free grammars. Acceptance within such a grammar can be determined by regular expressions or finite state automata. They are more restrictive than CFGs, but this reduction in power is a trade-off against the ease of determining whether a sentence is valid within a language defined by a regular grammar. Production rules involve a single non-terminal which may be transformed into one of the following: a single terminal, a single non-terminal and a single terminal, or the empty string.

Backus-Naur Form

Backus-Naur Form (BNF) is a notation used to formally describe a (context free) grammar for a language [56]. By describing a grammar in this manner, errors and ambiguity can be avoided. Grammars defined in BNF (or a variant like Extended BNF (EBNF), ISO/IEC standard 14977:1996) are used to clearly establish the language represented; syntax and content can be distinguished. The simple format of BNF, combined with the unambiguity compared to a verbose textual description of the language, allow the grammar to be read

[1]The author chooses not to opine on the topic of mathematical realism or constructivism, with regards to the creation or identification of mathematics

programmatically. Since a grammar can now be processed by a computer, parser generators using a standard format are possible - indeed YACC takes such an input.

3.2.2 Parsing algorithms

Parsing algorithms can generally be categorised as either top-down or bottom-up approaches. These terms are briefly elaborated on and the relevance to this project explained.

Lookahead Lookahead is an important concept in parsing, referring to the number of tokens ahead of that which is being processed currently to make a decision as to which rule should be applied. By minimising the lookahead required, the algorithm can be made considerably more efficient. For example, ANTLR can construct LL(k) parsers, which can vary the lookahead required to achieve the desired trade-off. The number of lookahead tokens required before backtracking must occur is often denoted as a positive integer in parentheses following the description of the parser e.g. LL(1).

Top-down parsing

Top-down parsing attempts to apply the production rules to the start symbol (and subsequent non-terminal symbols which are introduced by the application of the rules) to produce the input to the parser. LL parsers use the top-down approach for context free grammars. Input is read from left to right and derivations are executed on the leftmost non-terminal first.

Typical algorithms make use of a stack in their implementation onto which unprocessed terminals and non-terminals are pushed. A parsing table can deduce if some number of elements on the top of the stack can be popped off which is the case when these elements match a rule of the grammar.

Bottom-up parsing

Bottom-up parsing takes the input to the parser and attempts to match the production rules in reverse, such that non-terminals are introduced until the start symbol is reached. LR parsers use the bottom-up approach for context free grammars. Similarly to LL parsers, input is read from left to right however the rightmost non-terminal is replaced first (a rightmost derivation).

Shift-reduce is a common technique used in LR parsers. Given some input token, the parser can deduce which action should be applied based on the current state of the system; this decision process is encoded into a parse table, indicating how to proceed. An input token can cause the parser to shift, reduce, accept or error. A shift causes the token to be placed onto the stack. Reduction involves a number of elements on the top of the stack which match a rule to be popped from the stack. A non-terminal is pushed onto the stack; this

replaces the set of elements removed from the stack which match the right hand side of a rule, with the corresponding symbol on the left side of the same rule.

3.2.3 YACC/GNU Bison & Lex/Flex

GNU Bison (henceforth referred to simply as Bison) is an open source version of YACC, a popular parser generator used on the UNIX operating system. Since Bison is compatible with most aspects of YACC [57], as well as offering a number of improvements, we shall be using this parser generator to generate a parser for the system. flex is a lexical analyser generator (lexer) which is a free software alternative to the "original" lex and is largely compatible. This tool is used to generate a lexer; itself a tool to "recognise lexical patterns in text" [58].

These tools have been selected for this project because the author has experience using them and the availability of support due to their widespread use. An ANSI C lex specification [59] and YACC grammar [60] are available and shall be used within this project.

An input specification consists of a number of regular expressions and associated rules written in C to be executed when the regular expression is matched against the next piece of input text. These rules are combined into a function called yylex in the resultant lexer generated by the tool. The input file to the lexer is read and the longest block is sought which matches a pattern. The rule corresponding to the token matched is then executed; a typical rule in a lexer used in the frontend of a compiler would return the type of the token encountered for any keywords and create a record (or otherwise wrap/store) any other values encountered, such as integers and identifiers.

flex outputs a lexer containing the yylex function in lex.yy.c by default, along with all other internals required for token recognition and execution. It is not uncommon, especially in compiler frontends, for lex/flex to be used with YACC/Bison, the latter of which provides a parser operating on tokens provided by the lexer. YACC based parsers expect to call a function called yylex to receive the next input token, which is subsequently matched against the parser rules

The ANSI C lex specification was used with minimal modification. It was necessary to provide a more complete function which could accurately determine whether an input which matched that of an identifier was an identifier or a typename (defined by a **typedef** statement). A considerable amount of modification was required for the YACC specification since only the grammar rules were provided - actions for each rule to build the tree and perform any analysis must be added by the programmer.

Parsing algorithms used

Parsers generated by YACC/Bison are LALR parsers and use a bottom-up approach. Care must be taken to reduce or remove any conflicts which may arise in the grammar which may affect the resultant parser. All other algorithms used within the project use a top-down

(recursive) approach to perform various tasks on the parse tree.

3.3 Program transformation

Program transformations are used for a number of quite different purposes but generally a transformation converts an existing source of information via some sequence of operations to produce a new work in a potentially different language, which meets the aims of the transformation. Examples and a brief summary of different types of transformations are discussed below [61].

Program transformations can be split into two types - *translations* in which the source and target language are different and *rephrasings* which have the same language. Translations have an effect on the level of abstraction - some examples are listed below:

- Program synthesis. The level of abstraction of an input is lowered. In software development, compilation is an example of this transformation.
- Program migration. The level of abstraction is maintained, but the language changes. An example would be porting an application to another platform or programming language.
- Reverse engineering. The level of abstraction of an input is raised. A higher level version of the input is produced, e.g. decompilation.
- Program analysis. The level of abstraction is not directly considered. Instead, a subset of the input language and source is extracted or emphasised. An example of program analysis would be extracting the control flow of an application from the source code.

For completeness, a list of example rephrasings is provided. These transformations produce a different program in the same language - these have the same meaning and same language, but may use different structures/sentences to improve expression, clarity or enhance some feature/aspect.

- Program normalisation. Reduce a program to some subset, thereby reducing the syntactic complexity, e.g. simplification & canonicalisation.
- Program optimisation. Improve run-time memory or time requirements of a program, e.g. inlining, common subexpression elimination & constant folding.
- Program refactoring. Restructuring of a program to enhance clarity.
- Program reflection. Change program such that it also computes something about itself, e.g. trace execution.

- Software renovation. Fixing a bug in a program such that new requirements are met, e.g. introduction of a new currency.

According to the list of transformations, this project is a hybrid between "program migration" and "program analysis". The language changes from standard C to stackless C while maintaining almost identical semantics and level of abstraction. Program analysis is used by the system to highlight the use of the stack and to translate the input source code to a version which minimises stack usage.

3.4 C

Originally the creation of Dennis Ritchie [62] whilst working at Bell Labs in the late 1960s and early 1970s, C has become one of the most widely used programming languages particularly in the systems programming area. Some brief details are given for relevant sections of the language which will be of importance in the design and implementation of the system.

There are a number of different token types in C. These are: comments, identifiers, keywords, constants (integer, character, float) and strings. Also found in C are arrays, functions, pointers and structures. The program transformation will generally be at the statement and sub-statement level - it will not be sufficient to change individual tokens, but nor will it be necessary to make use of several advanced aspects of the functionality of C.

3.4.1 Scopes & variable declaration keywords

This section briefly covers scoping in C and the set of keywords which can be applied to variable declarations. It is important to mention and discuss these concepts so their impact on the transformations can be understood. Some keywords may not permitted in the proposed solution or may simply pose a challenge to transform (see section 4.5.1).

There is a direct relation between the location of a stack allocated object and scope - activation records are used to store objects on the stack for a block of code. This is important for the transformations required to introduce a software-controlled, heap-allocated stack. Similarly, the static scoping that exists in C must be maintained in the transformed code.

The following scopes exist in C [63, 64]:

Block scope (i.e. within braces). Any variables declared within such a block are visible from their declaration until the block ends. This is a form of local scope, and similarly, a variable in such a block is a local variable.

Nested scope Block scopes can also be nested, with the innermost block taking priority for variable lookup/resolution.

Function scope exists within an active function. This is limited to the goto statement.

Program scope or global scope covers any variables declared outside of any other scope. This can be restricted to the current file using the **static** keyword for a global variable.

A class of keywords known as storage class specifiers can also be associated with variable declarations to control the temporal aspect of a variable (whereas the aforementioned scope rules define the spaciality of the variable).

auto The location of this variable is temporary. This keyword is used infrequently since this it is otherwise implied.

static When not applied to a variable in the global scope, this keyword indicates that the value of a variable persists between calls to the function in which the variable is declared.

register Used for optimisation purposes, to signal to the compiler that a variable is best stored within a register rather than memory.

extern Specifies that a variable used within a file refers to a global variable within the program scope (and outside of the file scope), i.e. there exists a global variable definition in another file to which this variable refers to and should be linked.

Further to the previous set of keywords, the following are storage class modifiers and alter how the value of a variable may be manipulated:

const Indicates that the value of the variable cannot be changed post-initialisation.

volatile Another keyword passed to the compiler which signals that a variable may be modified by non-direct assignment means.

These specifiers will prove to be important when designing the transformations and understanding which keywords can still be applied to the transformed variable declarations.

3.4.2 Problems

Identifier/typename ambiguity

A context free grammar can be written for the language of C, permitting the use of many widely used tools which can generate parsers with efficient algorithms. There is however, a problem with accurately parsing C source code in one pass and without any external context, related to the ambiguity between an identifier and a type (namely those which arise from a **typedef** statement). The ambiguity is at the language level, and care must be taken when parsing - a bottom-up parsing approach would produce a reduce-reduce conflict. The following example (pre-C99) attempts to highlight the ambiguity:

Listing 3.1: Identifier/typename ambiguity example

```
foo(my_bool, my_uint, my_string);
```

If listing 3.1 is considered to be a function declaration, the function name is foo with implicit return type, and accepts three parameters of types my_bool, my_uint and my_string. Alternatively, this could be a function call to foo with three variables with identifiers my_bool, my_uint and my_string being passed as parameters.

C PreProcessor (CPP)

The C PreProcessor is an important aspect of C. While not part of the C language, it is part of the C specification [64]. While any input source code is expected to contain preprocessor directives, these will not be recognised by the transformation system. It is a simple design decision to omit support for the CPP: the focus of the transformations being implemented are to operate on input written in the C language itself. Some limited support could be introduced to merely recognise a block of text as a preprocessor directive, and to output it verbatim in the resultant source code. This directive may however, be a macro [65] which will otherwise interfere with or alter the transformations applied. Therefore, C preprocessor support is deemed out of scope for this investigation. For a full implementation, an existing implementation would be reused.

Most compilers will be able to interpret preprocessor directives independently of the lexical and semantic analysis of the C code itself. The resultant code could be used as an input and would consist of just C, bypassing the restriction imposed by this system which is a toolchain issue.

3.5 Summary

A considerable amount of this project revolves around parsing. This section describes how the concepts fit together as part of the system.

A lexer is required to break up the input source code into the various tokens which make up C and it's syntax. A lexer is generated from a lexer specification, consisting of an ordered list of regular expressions which represent text to be recognised as a token. Using a tool such as flex, a lexer can be generated for the given specification.

These tokens can be used by the parser to match rules which dictate what is and what is not valid C code. A grammar file consisting of rules written in BNF and a corresponding set of actions is used as in input for bison, which generates a parser which attempts to match input tokens against the rules. Parsers generated by YACC-like tools are LALR (LookAhead LR) and use a bottom-up parsing technique. This type of parser is more powerful than LR(0) parsers, yet remains efficient in terms of the size of the resultant parser compared to the number of possible grammars supported. Lookahead in an LALR

parser is flexible and provides this trade-off since the lookahead set is a smaller set of symbols which are associated with the current state and context of the parser.

There are few C grammars which are freely available online for any reasonable parsing tools. Two very similar YACC/GNU Bison grammars exist which claim to parse C89/C90; both are essentially the same in structure and completeness (LINKS). These grammars have been used as the basis of variants constructed for other tools and parser generators (e.g. ANTLR). Grammar ambiguity is not a problem in this grammar, however there are other parsing issues (identifier/typename ambiguity, section 3.4.2).

All other "parsing techniques" used in this system will be recursive (i.e. a top-down approach), such as annotating the parse tree using a tree walk. The tree will be walked repeatedly as part of the semantic analysis, transformation and code generation stages.

Chapter 4

Proposed Solution

This chapter deals with the requirements of the system, some of the approaches available to fulfil the requirements and various implementation choices for the chosen techniques.

4.1 Requirements

Since the tool produced is a source to source transformation system, it is only the resultant source code which can be referred to as “stackless”. The choice of compiler, e.g. GCC (compiler choice is discussed later in this section - section 4.4.6) will have a considerable impact on how the machine code is generated, and to what extent the stack is used.

This proof of concept implementation will only be able to process a single input source file, passed in via standard input. References to code in external files, ranging from functions in linked library files to the contents of header files which comprise a programming project, may not be resolved. This limitation has been covered in section 3.4.2 and also in chapter 6.

It should be noted that due to the nature of this tool and the C programming language, removing the stack entirely from an application to produce a truly stackless executable (independent of compiler etc.) is not possible at the source code level. The application is designed to minimise the use of the stack for code within the processed file.

4.1.1 Functional requirements

The system should conform to the aims and objectives of the project of investigating and implementing techniques to perform transformations to stackless C. Ideally all valid C will be handled, however as a proof of concept implementation, it is likely that a number of more advanced or complicated aspects of the C language will not be supported as well.

4.1.2 Interface & software requirements

The system takes an input of (valid) C source code and outputs valid C source code. No user interaction is required, so a GUI is not required and I/O can be performed via standard input and output from the command line. The tool will be developed on Linux, but should compile on other operating systems with minimal changes.

4.1.3 Non-functional requirements

The system should produce standards compliant code which does not introduce bugs. The system should perform the transformations in a reasonably short amount of time on a reasonably small amount of resources - the transformations should be applied within a matter of seconds on an average desktop computer.

4.1.4 Data stackless requirements

Automatic variables are variables which are local to a scope, stored on the stack and not declared **static**. This means that space must be reserved within the activation record of the corresponding scope for the value of a variable to be stored; this activation record is automatically removed from the stack at the end of the scope block, thereby destroying the value associated with the variable. The **auto** keyword is inferred for such variables, when no storage class specifier is otherwise specified. The term *data frame* will refer to the variables stored within an activation record, and contains no data related to control flow (such as the return address).

When parsing the input source code, it will be important to ensure that scoping information is well-maintained because each scope will have a corresponding structure defined. In the case of an empty scope, i.e. a scope which exists but has no variables defined within it, a dummy structure can be used instead. Once the input has been tokenised and parsed, semantic analysis will be required and information accumulated in structures about variables, functions and similar.

Structures for each scope can be built, with each automatic variable in a scope mapping to a field of the same name in the structure representing the current scope. Once the structures are generated, the references to these variables and the use of parameters to functions can be transformed.

4.1.5 Control flow stackless requirements

Traditionally, a function call would grow the stack by the required amount for the activation record of the function called to sit at the top of the stack. Within this activation record, the current instruction pointer (and registers depending on the architecture) would be

saved. Execution can then continue from the new instruction pointer at the beginning of the function body.

Once the function body has been executed (and/or a return statement is encountered), control must return to the previous scope, i.e. the instruction after the instruction pointer stored in the activation record - the return address. The stack is shrunk by popping the activation record for the called function. Execution then resumes in the original scope.

To achieve a stackless nature in terms of control flow, the core idea of the traditional control flow must be preserved - control jumps to another point in the code, executes for some number of operations and then execution continues after the initial function call. Aside from the name of the function to call (i.e. the location to jump to), the return address is the important piece of data which must be stored. It may be necessary to make use of C standard library functionality to store the current state of the system before the jump is made. This operation would store the values of the registers, which could be restored if needed to resume the application in the state before the function call was made.

Mimicking the traditional approach will be necessary to produce a reasonable replacement for function calls and minimising the stack usage. Making use of language extensions appears to be the only clean way of implementing this aspect of the project. The approach that will be taken is architecture specific, but should not be difficult to adapt to run on other systems through the clever use of **#define** statements in the C preprocessor (although this is not covered in this proof of concept implementation).

4.2 Modelling the stack

The C programming language standard does not dictate that a compiler should produce machine code which makes use of a stack when compiling C source code. There is no intrinsic link between C and the use of stacks, however the use of stacks in the execution of these programs is by far the most common method used (since it is relatively simple, supported in hardware and allows recursion and non-static variables, as discussed in section 2.1.2).

Instead, the "stack" can be modelled on the heap and is software-managed within the program (i.e. through the source code), rather than hardware-managed (and using machine code to manipulate the stack) as is typical for compilers such as GCC running on the x86 architecture.

Further to the notion of a heap-allocated stack, there is the opportunity to divide the data that is normally placed onto the stack (i.e. activation records containing local variables and a return address) into two sub-stacks. Each sub-stack would act in a typical LIFO[1] manner, with data frames storing blocks of local variables and call activation records storing the state of the application and the return address. This is shown by figure 4.1 and listing 4.1 below - the code listing shows some of the code which must be added and serves as an

[1] Last In, First Out

example to reinforce the figure.

Listing 4.1: Storage of heap-allocated stack

```
typedef struct ... { ... } *GlobalStack;

GlobalStack global_stack;        // Pointer (stored on the stack) to
    heap allocated structure

int main() {
        global_stack = initialise_global_stack();        // Allocated
            on the heap
        ...
}
```

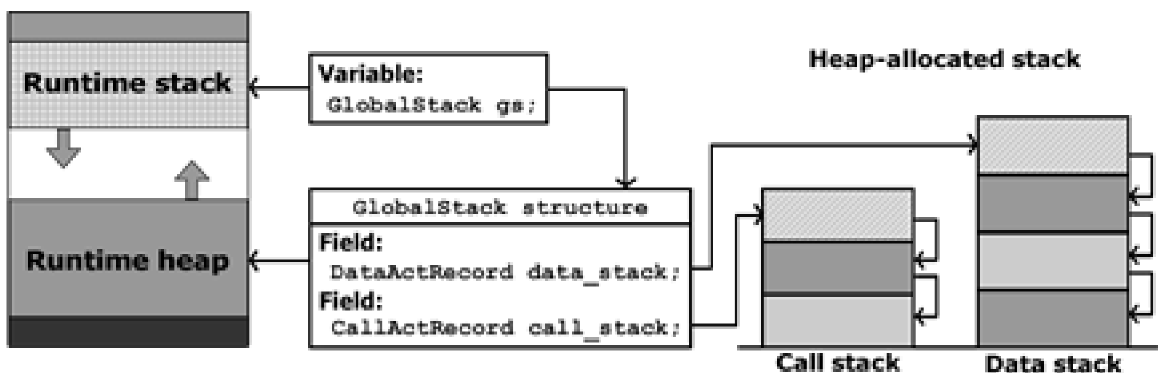

Figure 4.1: Modelling the call & data stack on the heap

4.2.1 Separation of control flow & program data

Buffer overflows are a significant problem in programs written in languages such as C. A malicious user may note the existence of a buffer overflow and exploit it to gain control over an application, which in turn may compromise the entire system (depending on the application compromised). Most buffer overflows require the attacker to modify the control flow of a running application. This is achieved by exploiting the combination of control flow data and variables on the stack - by writing past the end of an array, i.e. a buffer, the return address can be modified.

Separation of control flow and program data is an effective technique for preventing buffer overflow attacks, thereby reducing or hindering attack vectors that can be used against an application. The primary reasons for combining control flow and program data together are associated with convenience and locality. It seems both sensible and reasonable to maintain all data related to an event in a single block; this corresponds to the return address & local variables associated with a block or function call. However, this gives rise to the buffer overflow vulnerability, so splitting the stack into two sub-stacks based on the type of information stored will ensure that "execution" operations only occur on the call

sub-stack and "write" operations on the program data sub-stack. In effect, this produces a similar effect to that of the W^X system, used to protect against write and execution operations being carried out on essentially the same data.

A side effect of dynamically allocating activation records & data frames for the stack is the absence of the guarantee that existed for a traditional stack of contiguity. A more traditional stack grows linearly and contiguously, therefore any buffer overflows or similar writes to the stack which do not maintain the integrity of surrounding data, could potentially overwrite a large amount of data. If a guarantee that memory is randomly allocated, ASLR (introduced in section 2.3.1) should be used.

Canaries Canaries can be used to prevent and/or detect such corruption (discussed earlier in section 2.3.1). These small pieces of data are placed around blocks of data on the stack, such as around control flow data, or around fields used to store arrays. If the value of the canary has been changed then there has been corruption of data on the stack. The separation of control flow and program data removes a "use case" for canaries. That is not to render canaries obsolete, but potentially less important or relevant to fewer circumstances.

As a dynamic memory allocation can return a block of memory of the appropriate size from anywhere on the heap, there is no guarantee that the current data frame directly follows on/succeeds the previous data frame. This may limit undetected, subtle and/or non-fatal overflow damage/corruption to a smaller block - still problematic, but potentially less severe.

4.3 Options for heap allocation of data

4.3.1 Options for heap allocation of program data

Defining structures for each scope and allocating instances on the heap provides a relatively simple method of obtaining the correct amount of memory (i.e. sufficient for all variables in the scope) and enabling easy load/store of values. This is possible because the variables declared in a scope are known at compile-time.

An alternative solution, which would be considerably more low-level and difficult to implement, is to allocate one block of memory and extract values from it based on the length & offset of the variable type. For example, all variables of type **char** could be grouped together and allocated in one large block. Each byte in the block allocated could store a character, although this solution does not scale well for a number of variable types. Extracting an integer from such a block would require four bytes to be read and the values in each combined to produce a single value.

Instead, each variable declaration could be replaced with a declaration and malloc call. While the values of these variables could then be stored on the heap, the variable itself (a

pointer) would still be stored on the stack and therefore the aim of the project is barely met. In a large number of cases, this would substantially increase the memory usage of the program since a pointer *and* space on the heap would be required for every variable.

For this system, the first technique will be implemented because it is an achievable solution which provides a satisfactory approach to solving the problem of storing stack allocated data on the heap.

4.3.2 Options for heap allocation of control flow information

Within standards compliant C, the most obvious method of removing stack overhead (particularly with respect to return addresses in activation records) would be to make the program monolithic, with control flow managed through the source code annotated with goto statements and labels. This would require there to be just one function and would pose difficulties in passing arguments around and ensuring that variable scoping is maintained. The use of **goto** statements to jump between labels in different functions is not permitted because a **goto** statement cannot jump between functions (as described by Jones [64], section 6.8.6.1).

A number of variants are proposed instead:

Save & restore of instruction pointer Inline assembly is supported by GCC. It is possible to extract the current instruction pointer (IP) value from it's register; the discussion shall continue with regards to the x86 architecture. Unfortunately, there is no simple method of finding the current IP value which is stored in a special register - EIP. A call to a dummy function must be made, which pushes the current IP onto the stack to use as the return value when returning control to the caller. By popping the stack and copying the register value, the value in EIP can be obtained, which is the location at which the call statement was made. Listing 4.2 shows how the instruction pointer can be extracted.

Listing 4.2: Extraction of EIP value

```
unsigned long eip;
asm("call get_eip; get_eip: pop %%eax; movl %%eax,
    %0":"=r"(eip)::"%eax");
```

By storing an incremented value of the IP and then doing an (unconditional) jump to it, a simple call-return system has been implemented. An increment must be applied to the IP value to ensure that control returns to the correct point in the caller - the value in EIP must be obtained and subsequently stored on the call stack prior to the function being "called". This increment is required because the value obtained refers to the location of the instruction which actually obtained the value, and not to the desired return address. Once the value is obtained, it must be stored on the call stack, then the function call must be issued and control should return to the instruction immediately after the function call. From preliminary testing, calculating the increment required may not be possible.

The number of instructions which must be "skipped" to reach the desired return address varies between programs (and is obviously architecture dependent). The required value can be deduced by compiling the input source and investigating the machine code produced, however this is not practical (or perhaps even feasible) in the transformation system.

Save & restore of line number Using a combination of the above method and a technique used in Protothreads [38], unique labels can be generated which include the line number via the C preprocessor (using the __LINE__ macro). Storing the return address equates to storing the line number to return to, or where the function was called. Unfortunately, this approach loses some of the strengths and simplicity which are present in the Protothreads implementation because this method must be added to the code, whereas in Protothreads, the programmer writes code using the macro functions provided. This difference means that it is harder to obtain the correct location i.e. the line number which denotes the label to jump to, therefore this technique is not seen as being particularly flexible or effective.

A label-based call mechanism The aforementioned technique of removing the scopes associated with a function is not a particularly clean solution. It would reduce an application down to a single function, main(), with the original functions separated by labels. Removing the function definitions would be necessary for an approach that used just goto statements and labels because one cannot jump to a label outside of the current function using a goto statement. Due to the unattractiveness of this option, it shall not be considered or discussed further.

An inline-label-based call mechanism GCC's inline assembly could be used to place an "inline label" at the beginning of each function. By inserting a label in assembly at the beginning of the function, control can jump to the beginning of a function (without any explicit function call) and more importantly, without the restrictions surrounding the use of goto statements passing control to a different scope. Rather than using goto statements to change the control flow, inline assembly statements can be used to unconditionally jump to either the start of a function to be called, or jump back to the appropriate position in the caller body. This makes use of architecture specific assembly, however substituting the appropriate unconditional jump instruction for the target architecture would not be a difficult extension to the project.

Save & restore of label address Another extension provided by GCC allows the address of a label to be taken using the "&&" unary operator. A label should be inserted immediately after each function call i.e. after the jump to the callee function and before any processing of the return value is completed. Provided that every such label has a unique name, the address of the inserted label can be used as the return address and pushed onto the call stack.

In all of the cases outlined above, control can be resumed in the caller function (when the callee function has finished executing), by popping the return address from the stack and jumping to it.

The solution that will be used in this system will be a combination of the last two mechanisms. Inline labels will be inserted at the beginning of each function body to provide a location to jump to. Standard labels will be inserted after each function call and the address of each label stored on the call stack. An issue which will require further investigation is the use of registers when the code is compiled; values held in registers before a jump may need to be saved, otherwise the program will be left in an inconsistent state (thereby affecting the stability of the compiled program). In this circumstance, the value of the registers may need to be saved. This could be done "manually", through the use of assembly code to extract and store the values in a custom-defined structure; alternatively, an existing structure used for advanced control flow patterns such as those defined in the setcontext or setjmp family of library headers could be used.

Naturally, when using inline assembly, the code produced will no longer be compiler or architecture independent. Restricting the compiler in this way is not seen as being problematic. Dependence on a specific architecture could be avoided through either the use of the preprocessor or a command line argument to determine the target architecture and embed the appropriate assembly.

4.4 Implementation choices

A number of implementation issues are briefly addressed below, providing decisions and justifications for the chosen options.

4.4.1 A fixed stack size?

Stack overflow errors can occur when a program exceeds the size of stack memory which is (statically) allocated when the program initialises. Having a stack with a fixed upper limit will result in a semi-arbitrary limitation being imposed on the program itself. A stack overflow is most likely to occur with uncontrolled recursion - each time a non-tail call optimised recursive function call is made, another activation record is placed onto the stack to store the return address and any variables required.

Such overflow problems can be avoided if the stack is dynamically allocated. This is not to say that the dynamic(-ally allocated) stack is infallible; the machine may still run out of memory or otherwise be unable to heap allocate an activation record to place onto the stack. The solution is however, better in terms of flexibility.

4.4.2 Contiguous or fragmented stack?

For a dynamically allocated stack, there are two main methods of implementation - a linked list or an array. A linked list will allow a chain of dynamically allocated nodes and requires just a single pointer to the top of the list. Using this method, the data and the meta-data for the data structure are stored within the node, but the stack is fragmented throughout memory.

An array would require a contiguous block of memory which would store an activation record in each index. It would be necessary to maintain a counter for the top index of the array (unless every array element was shifted to ensure that the top element was always in index zero - a horribly inefficient implementation choice).

Pros and cons The primary advantage of a linked list is that there is no arbitrary upper bound imposed on the number of elements which can appear in the list. This is countered by the relative clumsiness of referring to an activation record/data frame which is not on the top of the stack because the links/pointers to the previous elements must be traversed until the required node is reached.

For an array, the opposite is the case. An upper limit is imposed on the number of elements which can be on the stack. However, since the scope of a variable is known at compile-time, the index corresponding to the location of the data frame on the stack can be used, which greatly simplifies lookup of variables which are stored in a data frame other than that at the top of the stack.

4.4.3 Garbage collection or free()?

Garbage collection is not required by virtue of the design or implementation, but shall be used in the transformed source code. Since all stack data which would normally be cleared up "behind the scenes" is now stored on the heap, the activation records & data frames placed onto the heap allocated stack must be freed at an appropriate point. While it is possible to insert calls to free memory at appropriate points in the transformed source code because the scope of variables is known, the implementation and resultant code become more complicated. There is little advantage obtained in this case for manually freeing dynamically allocated memory when garbage collection can be used. Furthermore, effort can be dedicated to the implementation and optimisation of the transformations, rather than further tree manipulations.

4.4.4 Binary tree or n-ary tree?

A binary tree is produced by the parser to represent the parsed input source code. The choice to use a binary tree rather than an n-ary tree or have a variable number of child nodes was made for simplicity in terms of constructing and processing the tree. Each node

has an integer type and two child node pointers; a medley of flags and other pointers store auxiliary information such as the environment structure for this scope and whether a node should be processed further in later stages of the transformation.

4.4.5 Singly or doubly-linked tree?

A singly linked tree is produced by the parser. Any given node can have a type and 0-2 children links; there is no pointer to the parent node. This choice was made to simplify the creation and modification of nodes, although with hindsight, a “parent link” would have been useful; this is discussed later in improvements (section 6.3.3).

4.4.6 Compiler usage

GCC has been selected as the compiler for this project for a number of reasons:

- GCC is a widely used, freely available, robust and effective C compiler.
- The code generated will require the use of extensions to the C programming language [66].
- By selecting a compiler that will be used with the output source code, testing and demonstration of results can be standardised.

4.5 Transformations required

The transformations required can be numerous in some cases, however for many aspects of C, the code requires little to no modification. A non-exhaustive list of the transformations required is presented below:

- Basic variable types can map directly.
- Variable definitions are moved to a structure and removed from their existing scope. These new structures should be created for each scope (global, function parameter, function body & nested block) and inserted after all existing structures.
- Variable initialisations (i.e. at time of definition) must be extracted from the definition. Global variable initialisations must be completed early in the programs execution, i.e. before they are used. A reasonable solution would be to move the initialisations to the beginning of the main function.
- Variable usage is replaced with the appropriate location on the dynamically allocated heap.

- Existing structures & type definitions should map directly.
- Storage duration & type qualifiers may cause problems (see section 4.5.1).
- Functions have no return value (return type of **void**).
- Functions have no parameters.
- Function calls must be removed. Parameters are loaded onto the heap and the return address is stored.
- Labels must be added at the beginning of each function body and immediately after each function call.
- Return statements load the return value into the structure on the data stack and restore control to the address on the top of the call stack.

4.5.1 Problems & limitations

Keyword support

There are some keywords which will have to be stripped from the input source code. For example, declaring a field in a structure **static** would not be valid. A variable declaration combined with initialisation is transformed into two statements - the declaration of a field in a structure and then an assignment of a value to this field. This cannot be mapped to a stackless version if the variable was declared constant.

In other cases, it may be possible to implement a transformation which achieves the same effect as the input code, such as moving all static variables into a different scope and preserving values between function calls. However, aggregation of all global variables into a single scope may change the semantics of the resultant code.

Global variable declarations

As a result of what could be considered a "quirk" of the C specification, it is not required that all global variables will be defined at the top of a file and before any function definitions. This gives rise to a semantic difference between a function definition (a "function") which precedes a global variable declaration and a function which succeeds the same definition; the global variable is in the scope of the latter function because the function definition lies after the variable declaration, whereas the former function cannot access this variable. Listing 4.3 shows a code snippet which will not compile; move the global variable declaration and the program will compile. An interesting problem is introduced to the task of performing a program transformation with minimal semantic side effects, and one which will not be considered in the design and implementation of the data stackless transformation.

Listing 4.3: Example of change of semantics for global scope

```
#include <stdlib.h>
#include <stdio.h>

void foo() {
      global_int++;         // Not declared at this point
      return;
}

int global_int = 9;         // Declare global_int before foo() to compile

void bar() {
      printf("global_int = %d\n", global_int);
      return;
}

int main() { foo(); bar(); return EXIT_SUCCESS; }
```

Library functions

A significant limitation of this system is that it can only operate on the source code which is provided directly to the application. The most apparent effect of this is that functions within any libraries used or linked by the resultant compiled application will not be stackless in any way (unless, of course, the libraries have already been transformed). Producing a stackless library is rarely an option for a number of reasons. The source code for the library may not be available or convenient to recompile; issues regarding portability may arise; and dynamically linked libraries may be different (though not in terms of functionality) from system to system.

This situation is similar to that of other systems which attempt to use or change an aspect of the programming language. Cyclone [3, 67], a variant of C with the aim of producing safer code, also suffers from this problem. Some of the standard library functions & libraries have been rewritten to some extent to enable them to be used without issue in applications written in Cyclone. Therefore, a potential solution which partially fixes the problem, is to produce stackless variants of some core library functions by providing pre-transformed libraries and allow the user to include or link against these. Alternatively, the user could simply switch between calling conventions without issue.

Command line parameters

Command line parameters passed to the application, traditionally in "**int** argc" and "**char** **argv" will remain in the same place within the program memory. It is not feasible (or maybe impossible) to move these values onto the heap through a program transformation. This limitation has been identified and can be classed as a shortcoming in the transforma-

tion required due to the C specification and executable design, rather than a limitation of this particular implementation.

Transformation complexity

Transforming some elements of C will be particularly difficult because a simple mapping to a stackless version does not exist. Stack-based structure initialisation is syntactically simple, whereas a stackless implementation would require a heap based allocation followed by a series of assignments into the structure. This project will discover other similar issues as the transformations are developed and implemented.

It may be necessary to impose some restrictions or coding standards which are required of all input code. For example, a function of return type **void** may not have a return statement at the end of the function body. The complexity of transformations may rise considerably if more state information must be maintained. Every non-main function must return control to the function which called it. In the majority of cases, this will be indicated with a return statement, however if the return statement is omitted, scope information must be maintained to deduce where the stackless return statement(s) should be inserted. In this particular example, it would simplify the transformations if it was mandated that no implicit return statements are used. Alternatively, further semantic analysis would be required to patch the parse tree to include such "missing" statements to produce a normalised tree which the transformations can operate on.

Goto statements should work without issue since they do not change the stack. Care must be taken to analyse that control does not jump past any vital code inserted by the transformations, e.g. a goto statement should not bypass any code inserted for stackless return, which would otherwise leave the call sub-stack in an inconsistent state. The readability of the resultant code may be further reduced if goto statements and labels are in the input source because several new labels will also be added as part of the call stackless transformations.

Function pointers may be more problematic to implement since they will require a more powerful transformation. As part of the data stackless transformation, all functions are transformed to the same "type", i.e. "**void** function_name(**void**)". All references to function pointers must also be updated in this same manner. An additional problem arises when transforming function calls as part of the call stackless transformation. For a "normal" function call, the name of the function to be called is known, and therefore the name of the label inserted at the start of the corresponding function body can easily be obtained (by a simple lookup). A function pointer stores the address of a function, and therefore the name of the label which control must be transferred to cannot be looked up easily. This poses a problem for the current transformation design which must be overcome to support a larger proportion of C.

It is expected that a number of edge cases and more powerful & obscure aspects of the C specification may pose problems which have not been anticipated. These shall be

documented and solutions proposed where possible. Only a full implementation with complete testing can demonstrate that this tool will work without issue on all valid C code. Any defects in the grammar, lexer, tools used or problems which arise due to non standards compliant code are not considered at this stage.

4.6 Summary

This chapter has outlined the requirements of the system, which provide focus on the aims of the software implementation. Consideration has been given to a variety of techniques to achieve stacklessness in terms of program data and control flow through the use of transformations. The opportunities, benefits and problems associated with the suggested techniques and tree-based source-to-source transformation systems operating on and written in C are provided. To achieve program data stacklessness, each environment (or scope) will be modelled by a structure and these structures placed onto a heap-allocated, software-controlled data stack. For control flow stacklessness, function calls will be replaced with inline assembly unconditional jumps to labels inserted at the start of each function body, and each return statement will transfer control back to a label inserted after the function call in the caller.

Chapter 5

Implementation

This chapter will discuss the implementation details of the transformations based on the techniques selected in the previous chapter. A number of figures and code listings are included to provide some examples and context to support the explanations and justifications. Some details are given regarding auxiliary functionality (lexing, parsing), before covering the data stackless and control flow stackless methods. A discussion of limitations and issues, followed by a summary concludes this chapter.

5.1 Tree generation - lexing & parsing

As part of the frontend of the application, a lexer and parser are required to identify tokens and match these tokens against the rules of the grammar used. The lexer is generated by flex from an input specification [60], and the parser is generated by Bison from an input grammar provided in a BNF variant [59]. While the rules are supplied within the grammar file, the corresponding actions must be written to build the tree.

The lexer is relatively simple and wraps the token types and lexemes in a token structure. Identifiers, constants and string literals return a new token structure with both a lexeme and token type, whereas all other tokens produced just have a token type. Token types are defined by integers greater than 255 since this allows individual characters to use their corresponding ASCII value as a token type.

A string table is built by the lexer to identify symbols which have already been encountered. Each string has a corresponding integer value to simplify lookup and comparison of strings in the earlier stages of the transformation. This string table and the resultant parse tree are returned by the parser to the main program.

It is necessary for the lexer and parser to be linked, even if this does reduce coherence and increase coupling to some extent. C allows types to be defined through identifiers with the **typedef** keyword. A list of identifiers which are declared as type names must be maintained to enable the frontend of the application to distinguish between an identifier and a type

name which has been introduced through a **typedef** statement (see section 3.4.2).

The lexer is responsible for tokenising the input and providing a sequence of tokens to the parser. This includes identifying each lexeme and determining the corresponding token type. To distinguish between an identifier and a typename, the lexer requires a function to check whether the identifier has been previously defined as a typename. It is however, the responsibility of the parser to identify expressions and statements from the input token sequence and to start building some state and structure associated with the input. When a rule for a **typedef** statement is matched, the parser will set a flag to indicate that the identifier used should be added to the list of typenames. This list is shared between the parser and the lexer; the parser adds any and all identifiers which are to be recognised as types to the data structure, and the lexer calls a function when an identifier is found to determine whether the token type is a typename or identifier.

5.2 Data stackless method

5.2.1 Environments & semantic analysis

Scoping information is of particular importance because the stack allocated variables of each scope must be transformed to be stored on the heap. An environment structure is built via a tree walk over the tree for each scope. Whenever the scope changes (such as when a nested block is entered), a new structure is created; each scope has a type from an enumeration of "parameter scope", "function body scope" and "other nested block scope". A pointer in the new structure is set to point to the previous environment to produce a linked list of environments which can be traversed to find all symbols which are in a particular scope. The node in the parse tree which caused the change in scope stores a pointer to the new environment structure created, allowing the current environment to be tracked when walking the parse tree.

An internal structure representing a variable is created for each variable declaration encountered. A list of variables declared in a scope is built and added to the current environment.

Function definitions are also processed when building the environments. A structure representing a function definition is created and added to a list in the environment representing the global scope. This structure contains the name of the function, the return type and a list of variables which make up the formal parameters accepted by the function. Formal parameter data is required when transforming function calls as part of the data stackless transformation (see section 5.2.5) and the return type is required for building the structure which represents the parameters of the function (and is used to hold the return value, see section 5.2.2).

5.2.2 Structure construction

As all parameters and automatic variables are allocated on the stack, it is necessary to allocate and provide an effective load-store mechanism to replace the original declarations. All local variables for a particular scope are traditionally allocated on the stack within an activation record which is pushed and popped from the stack when the block is entered and exited respectively. The activation record can be software controlled and heap allocated by storing the program data that would otherwise be stored on the stack in a structure. A structure can be generated for each scope and heap allocated as and when it is required.

The list of environments built in the previous step (see section 5.2.1) is traversed. A structure is created for any environments which have one or more declarations or an environment which represents a parameter scope. The name of each structure is derived from the name of the function in which the current scope is defined with a unique identifier appended and this name is stored in the environment structure. A field is created for each variable declared in the current scope; the name and type of the variable are maintained, however other variable modifiers (as previously mentioned in section 4.5.1) may be lost since their presence is invalid within a structure. A structure defined for a function parameter scope has an extra "return value" field if the function for which it is defined has a non-**void** return type.

By storing the return value in the parameter scope, the caller function (which creates the parameter scope, loads all parameters and pushes the data frame onto the stack) will have access to the return value once the callee function has executed and returned. Using the variable identifier as the field identifier in the structure simplifies the transformation of loading and storing values.

Each structure also forms part of a **typedef** statement to simplify casting the structures when loading and saving values - a simple typename can replace "**struct** typename *". A further dummy structure is also defined. This structure has no fields and is used to represent environments which must be pushed onto the data stack but do not have any declarations and therefore require no fields. All structures generated in this step are directly after any existing global variables and structures.

Listing 5.1: Structures produced for a function definition

```
int foo(int x) {
        float f;
        double d;
        ...
        return x+3;
}

// The following structures are created for the function definition
    above:

// Structure to hold all parameters + return value of foo.
typedef struct foo_param_struct_ {
```

```
        int x;
        int __return_value;
} *fooParamStruct;

// Structure to hold all local variables declared in the body of foo.
typedef struct foo_body_struct_ {
        float f;
        double d;
} *fooBodyStruct;
```

5.2.3 The stack

The model of the stack on the heap is a structure which consists of two sub-stacks, i.e. a pointer to the top elements of both the "data stack" and "call stack". Both sub-stacks are singly linked lists storing the location of the element which was pushed onto these stacks previously (and will become the top element of the stack again when the current top element is popped off the stack).

The data stack is a linked list of data frame structures. This structure contains a "**void** *" field which stores the location of a heap allocated structure that accommodates all variables used in the scope that is to be entered (as shown in listing 5.2). A cast is necessary to extract any data stored in this data frame; the cast informs the program how to interpret the referenced structure which has no type information associated with it. See section 5.2.4 for loading and storing values.

Listing 5.2: Example use of **void*** in data frame

```
typedef struct data_frame_ {
        void                    * data;
        struct data_frame_      *prev;
} *DataFrame;

// Different types must be stored within a data frame on the data
    sub-stack.
typedef struct dummy_struct_ { } *dummyStruct;
typedef struct foo_body_struct_ { ... } *fooBodyStruct;
typedef struct foo_param_struct_ { ... } *fooParamStruct;
typedef struct bar_param_struct_ { ... } *barParamStruct;
```

In terms of compilation, C has the benefit of having static scoping which means that the location of a variable is known at compile time. This is important because the location of any variable on the software-controlled heap-allocated stack is known when the source transformation is applied, and so a compile (transformation) time lookup can be included. Substitution of strings is therefore possible without necessarily changing the semantics of the application.

The field name is known because it has the same identifier as the variable. Each variable belongs to a variable list in an environment which represents a scope, and each environment

has a structure to represent its contents. The name of the structure (i.e. the type of the structure containing a particular variable definition) is therefore determined via the environment in which the variable is located and the structure name is extracted. In the case that the variable is not in the current scope, the parent environment is checked by traversing the "previous" pointer; the number of traversals is equal to the offset of the activation record containing the variable from the top of the data stack.

Heap allocated stack initialisation A stack allocated variable in the global scope must be declared to store the location of the heap-allocated stack, as shown in listing 5.3. This variable must have global scope so that the software-controlled stack can be pushed or popped, read or written to from any point in the application. The structure itself is dynamically allocated, but requires a stack allocated position to store the location of the structure.

Listing 5.3: Initialisation of heap-allocated stack

```
typedef struct ... { } *GlobalStack;

GlobalStack global_stack;        // Global variable (pointer to heap
    allocated stack)

int main() {
        global_stack = initialise_global_stack();
        ...
}
```

The structure used to store the heap allocated stack(s) is defined in an external file. This file contains push, pop and initialisation functions for the stack structure and constructors for the data & call frame structures. The initialisation of the global variable for the heap allocated stack is the first statement within the main function because all other data is also heap allocated and referenced from this stack.

Listing 5.4: Original and transformed code setting up state shown in figure 5.1

```
// Original code. foo calls bar.
void foo() {
        int i = 9;
        double d;

        bar(5.6);
}

void bar(float f) {
        char c = 'a';
        int j;
        ...
}
```

```
// Transformed code. foo still calls bar.
void foo() {
        push_new_data_activation_record(global_stack,
           newDataActivationRecord(newFooBodyStruct()));
        ((foo_body)global_stack->data_stack->top->data)->i = 9;

        // call bar
        push_new_data_activation_record(global_stack,
           newDataActivationRecord(newBarParamStruct()));
        ((bar_param)global_stack->data_stack->top->data)->f = 5.6;
        bar();
        pop_data_activation_record(global_stack);

        // leave foo
        pop_data_activation_record(global_stack);
}

void bar() {
        push_new_data_activation_record(global_stack,
           newDataActivationRecord(newBarBodyStruct()));
        ((bar_body)global_stack->data_stack->top->data)->c = 'a';
        ...
        pop_data_activation_record(global_stack);
}
```

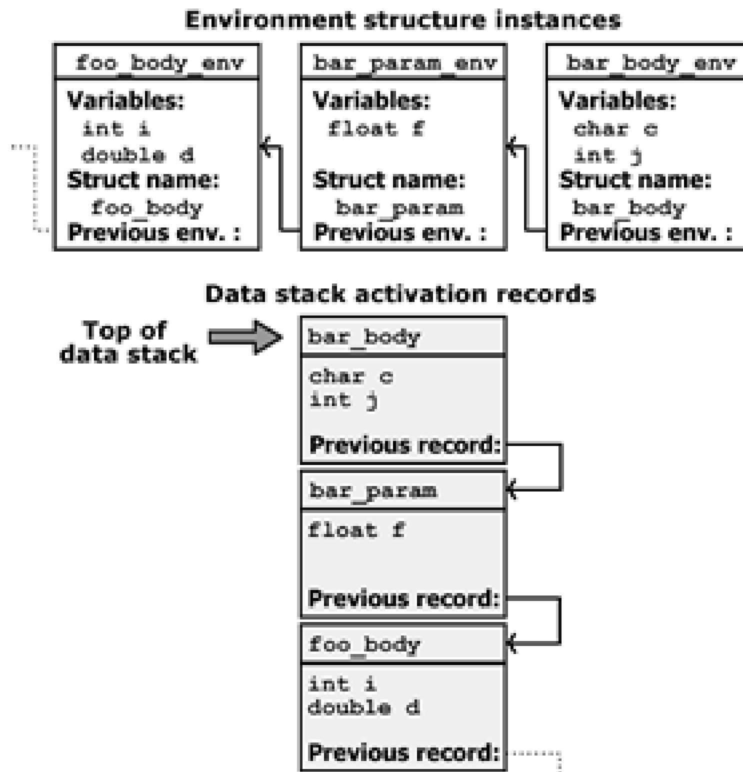

Figure 5.1: Mapping of environment structures to data frames

Figure 5.1 shows an example of the internal environment structures which represent the scopes that exist in the input source code and the corresponding elements on the data stack. Listing 5.4 shows the original and transformed code showing foo calling bar to establish the state shown in the figure. Each element on the data stack is a data frame which contains a pointer to a structure of type "struct name" as indicated in the corresponding environment structure. It is clearly shown that there is a direct mapping between the variables in an environment and the structure which is stored on the heap allocated data stack. For example, if there was a reference to "**float** f" and the data stack was as shown, this could be accessed via "((bar_param)global_stack−>data_stack−>top−>prev)−>f". This takes the second to top element of the stack, casts it to the appropriate type and then accesses the appropriate field. For comparison, "**char** c" and "**int** j" would be stored in "((bar_body)global_stack−>data_stack−>top)".

A note about casts Each data frame is implemented as a structure and stores a reference to a different structure defined for each scope. An instance of the appropriate structure is created whenever a scope is entered and these instances must be stored on the heap-allocated data stack. A data frame must therefore store an instance of any structure. Due to the typing system of C, no type information is stored and the value at the stored

memory address must be cast in order to access the data for this scope (see example in listing 5.5). The use of casts is possible because of the static scoping employed by C. When the transformations are executed, it is known which structure is in which location on the data substack and the appropriate cast can be inserted. Unfortunately, casts are a solution to a problem which is more elegantly handled in other languages, most notably the main object oriented languages. An object oriented approach may produce a cleaner design and implementation for this aspect of the program.

Listing 5.5: Cast required to access structures within data frames on data sub-stack

```
Access field 'f' in the second to top data frame of data sub-stack.
    'f' is in the fooParamStruct.
((fooBodyStruct)global_stack->data_stack->top->prev->data)->f;

// global_stack->data_stack->top gets the top data frame from the
    data_stack.
// The prev link goes to the previous element
// The void * data field is then cast to the correct type, and the
    field can be accessed
```

5.2.4 Loading & storing values

When a variable is used (which can generally be classified as either a load or store statement), it must be replaced with the appropriate heap based reference. This is achieved by walking the tree and tracking the current scope and corresponding environment structure. When a variable is encountered, its use can be replaced with the appropriate field in the activation record at the required offset from the top of the data stack. This can be established using a function call which looks up and returns the new heap location of the variable with respect to the current environment.

Within a function, there are at least two scopes currently active; the parameter and function body scope. This distinction is maintained in the transformed program - a structure is created for the parameters of a function and another for the function body, therefore within a function there are two structures on the data sub-stack for this function call (one for parameters and one for the body). The structures are similarly constructed with minor differences - formal parameter identifiers are used rather than local variable declarations and there is an additional return field with type corresponding to the return type of the function within the structure for the function parameters. These two structures could potentially be merged because no code can be placed between the parameter and function body scopes, however the distinction will be maintained for simplicity.

5.2.5 Function calls

A function call can be broken down into the following steps, each of which corresponds to a set of actions that comprise the transformation which must be applied:

- Evaluate any parameters.
- Load all parameters into registers and/or onto the stack.
- Call the function.
- Handle the return value, if any.
- Clear up the data pushed onto the stack.

Parameter evaluation & loading The values which are to be passed as the actual parameters to a function are either directly in scope of the caller function or must be evaluated. As a result, it is the caller's responsibility to create the data frame for the function call into which the parameters are loaded. Two data frames are put onto the data stack for a simple function call - one by the caller for the function parameters and return value (if any), and one by the callee for the local variables at the start of the function body.

A function call must match all actual parameters with the formal parameters specified in the function declaration. The structure representing the function being called is looked up from the global environment. All function parameters are stored as a list of variables, and the names of the formal parameters can be paired with each value in the actual parameter list. A sequence of statements are added to the source code:

- A function call to create the structure to hold the parameters. The name of the structure can be derived from the environment associated with the function being called.
- The heap allocated instance of the structure is wrapped in a new data frame and pushed onto the data stack.
- Each parameter is loaded by assigning the value to the corresponding field name in the structure just pushed onto the data stack, maintaining order. This process is comparable to that of loading & storing local variable values, but with the extra transformation complexity associated with matching formal and actual parameters.

In the case when there are no parameters for this function call, a dummy structure (an empty structure, as previously mentioned) wrapped in a data frame should be pushed onto the stack. This simplifies and standardises the process of pushing and popping data frames from the stack, albeit at the expense of a semi-redundant statement and the heap memory required. It is acknowledged that the solution implemented is not the most elegant, but it simplifies transformation.

If a parameter is a function call itself, a similar transformation must be executed. The parameter must be executed and the return value loaded into the appropriate field for the initial function being called. The complexity of the resultant code is slightly increased and the manipulation of the tree made quite difficult since several stackless function calls will occur in one block of code, each changing the state of the data stack.

Calling the function Any parameters listed in the function call must be removed from the function call as part of the transformation of functions. The function call is isolated and a parameter-free call to the function is made. See listing 5.6 for a code example.

Listing 5.6: Transformation of function call

```
// Function call abstracted from environment
...
foo(1,2,bar(3));

// The following would replace the function call:
...
push_new_data_activation_record(global_stack,
    newDataActivationRecord(newFooParamStruct()));
((FooParamStruct)global_stack->data_stack->top->data)->a = 1;
((FooParamStruct)global_stack->data_stack->top->data)->b = 2;

// --- Start call to bar ---
push_new_data_activation_record(global_stack,
    newDataActivationRecord(newBarParamStruct()));
((BarParamStruct)global_stack->data_stack->top->data)->d = 3;
bar();
((FooParamStruct)global_stack->data_stack->top->data)->c =
    ((BarParamStruct)global_stack->data_stack->top->data)->return_val;
pop_data_activation_record(global_stack);        // remove
    BarParamStruct
// --- End call to bar ---

foo();

// handle return value here
pop_data_activation_record(global_stack);        // remove
    FooParamStruct
```

Return values & clearup If the return value from the function call (if any) was used, i.e. as an operand for another operation, the function call should be replaced with the location of the return value. This is stored in the activation record on the top of the data stack, in the return value field, so a simple substitution can be made.

Once the return value has been used or stored (if at all), the activation record for the parameters passed to the callee function (and which stores the return value) should be popped from the data stack, thereby leaving the stack in the same state prior to the function call.

Listing 5.7: Transformation of function call and return value

```
void foobar() {
        int i = bar(7) + 8;
        ...
```

```
}

// The above function call will be replaced with the following:
void foobar() {
        push_data_activation_record(global_stack,
            newDataActivationRecord(newFoobarBodyStruct()));

        // Set up call to bar
        push_data_activation_record(global_stack,
            newDataActivationRecord(newBarParamStruct()));
        ((BarParamStruct)global_stack->data_stack->top->data)->d = 7;
        // Call bar & process return
        bar();

        ((FoobarBodyStruct)global_stack->data_stack->top->prev->data)->i
            = ((BarParamStruct)data_stack->top->data)->return_val + 8;
        pop_data_activation_record(global_stack); // remove
            BarParamStruct
        pop_data_activation_record(global_stack); // remove
            FoobarBodyStruct
        ...
}
```

Once the function call has returned, the return value expected (if any) can be read from the top activation record on the data stack (see listing 5.7 for code example). If the result of a function call was used in some manner, e.g. being used as an operand in a calculation, then it will be necessary to execute the function and then replace the call with the return value into the statement. Two calls to pop elements from the data stack should then be added to the source code - these are to remove the structure for the function body (which was added by the callee) and for the function parameters (added by the caller before the function call was issued).

5.2.6 Function declarations

Some relatively simple transformations must be applied to function calls which are listed and subsequently explained below:

- Whenever a new scope is entered, an instance of the structure representing the new scope must be pushed onto the data stack to store the new variables which must be stored. Similarly, when a nested-block scope is exited, the data stack should be popped to remove the top data frame.
- Any return statement which returns a value should load the value into the “return value” field in the parameter structure. For any return statement, data frames created since the function was entered should be popped from the data stack.
- Function calls are discussed in section 5.2.5.

- Use of variables is discussed in section 5.2.4.
- All parameters should be removed from any non-main function definition.
- The return type of all non-main function definitions should be set to **void**.

Listing 5.8: Transformation of function declarations

```
// Function declarations
void foo(int a, int b, int c);
int bar(int d);

// The following would replace the function declarations:
void foo();     // Parameters removed
void bar();     // Return type changed & parameters removed
```

The return type of all functions is converted to **void** because all return values are passed back in the structure storing the parameters of the called function; all return statements should also be removed, with the return value being loaded into the field in the structure. Similarly, all parameters can be removed since they are now passed to the callee in a structure put onto the heap-allocated stack by the caller. See listing 5.8 for a code example. The exception to these rules is the main function, which is required to be defined as **int** main() (with optional stack-based parameters "**int** argc" and "**char** **argv").

5.2.7 Global variables & initialisation

Similarly to all other scopes, variables in the global scope are aggregated into a structure of their own. An instance of this structure is pushed onto the data stack immediately after the heap allocated stack is initialised. It should be noted that global and static variables are not necessarily stored on the stack (or even the heap), but in a separate part of the program memory. Nonetheless, the transformations required are considered and applied for uniformity. It is important for this initialisation to be executed as early as possible since all existing global variables will be transformed to be stored on the heap allocated data stack.

Existing global variables are processed in one of two ways:

Declaration only All variable declarations are discovered and processed in the tree walk which is used to build the structures for each scope. Therefore global variable declarations are already mapped into the global structure and require no further processing; these declarations should be deleted from the program tree.

Declaration & initialisation Global variables which are both declared and initialised are broken down into the two constituent parts. The declaration has been handled previously in the same manner as expressed in the previous case. The initialisation of this

variable is moved to the beginning of the main function, appended to the two new statements. The order of initialisation should be maintained because the initialisation of a global variable may depend on other initialisations which precede it.

All global variables which have initialisations should now be loaded into the structure representing the global scope and all declarations removed.

The **static** keyword in C is a storage class specifier which has two meanings: when used within a nested block (such as a function), the value of the variable is persistent between calls to the function; otherwise the variable may only be accessed from the file in which it is declared rather than having global scope throughout the compiled files (and therefore application).

In the former case, a static variable would typically not be stored within an activation record for a function. Instead, static variables are stored below the heap in a statically sized section of read-write process memory. This can be mimicked in a transformed stackless variant by raising the static variables to the global scope such that they are stored in the structure which stores global variables. While this technically enlarges the scope of the variable, this is essentially what happens when the program is compiled anyway.

When a global variable is marked as static, this is restricting the visibility of the variable to this compilation unit. This is not supported within this implementation.

5.3 Control flow stackless method

A brief introduction to inline assembly as supported in GCC is given, since the chosen transformation will require the use of this extension. The limitations of C prevent it from being used throughout the entire transformation to minimise the use of the stack. How inline assembly contributes to the transformation and the individual steps of this phase are then discussed.

5.3.1 Inline assembly

GCC supports inline assembly code [68, 69, 70, 71]. This is assembly code using the AT&T UNIX syntax which is inserted verbatim into the code, much like a macro (and can subsequently be optimised). Inline assembly is important because it can interface with the C program via "extended ASM" which permits operands to be provided as well as instructions. Extended ASM simply extends the functionality of the asm keyword to permit a list of input & output variables and a list of registers which are used (and therefore the compiler should ensure that these registers do not contain unsaved values used by the rest of the program during the code generation phase). Specifying the **volatile** keyword will ensure that the compiler does not move the assembly statement as part of the optimisations performed; this may be of particular importance if the inserted statement has side effects.

It should be noted that there are several differences in syntax between the AT&T syntax

(used by GCC for inline assembly) and the more traditional Intel syntax. These shall not be covered in this report. An example of using GCC inline assembly can be found in listing 4.2.

Inline assembly would appear to be an effective technique to bypass the constraints imposed on control flow by the C specification, especially on the use of **goto** statements and changing scope [64]. The family of setjmp/longjmp functions allow control flow to jump between scopes. This mechanism cannot be used within a transformed system because the location must be saved to a structure representing the current state, which is restored later in the program execution. To issue a stackless function call for the first time, i.e. to transfer control to the body of another function, it would be necessary to restore a buffer (using longjmp) which represented control in the function being called. Clearly it is not possible for such a buffer to exist. Similarly, storing the current state in a buffer (using either the setjmp or ucontext family of functions), issuing a function call and then restoring the saved state would not be effective. Saving and restoring state in such a manner would cause a loop because the instruction pointer was also saved, rather than advanced past the instruction which issues the function call.

Inline assembly can be used to achieve non-local jumps - essentially an unrestricted **goto** statement. The issue of programming using goto statements as a form of control flow management is discouraged by Dijkstra [72]. In this situation there are no alternative control flow constructs and the use of jumps to manage control flow is introduced by the transformation, thereby relieving the burden of managing the control flow spaghetti which can arise through overuse of **goto** statements.

5.3.2 Transformations

The control flow stackless transformation is relatively simple to apply once the data stackless transformation has been completed. Post-data stackless transformation, function calls are isolated (i.e. they form an entire statement) and are without parameters (see listing 5.9). The following stages are implemented to meet the requirements of: knowing the location to transfer control to, jumping to this location and having a stored location to which control can be returned.

Listing 5.9: Function call post-data stackless transformation

```
...
int i = foo(4);
...

// Post data stackless transformation becomes:
...
push_data_activation_record(global_stack,
    newDataActivationRecord(newFooParamStruct()));
((foo_param)global_stack->data_stack->top->data)->a = 4;
foo();  // 'Isolated' function call
```

```
((current_body)global_stack->data_stack->top->prev->data)->i =
    ((foo_param)global_stack->data_stack->top->data)->__return_value;
    // Assignment of return value to i
pop_data_activation_record(global_stack);         // Remove foo
    structure from data stack
...
```

Each function must have an inline assembly label inserted as the first statement of the function body. This provides a known fixed location which control can be transferred to when entering a function. Inserting the label node into each function is achieved by iterating over the list of function definitions stored in the global environment, which contain a pointer to the function body. The label cannot be a standard C label since the use of goto statements is restricted in C (as described by Jones [64], section 6.8.6.1 and previously mentioned in section 4.3.2).

Subsequently, each function call must be replaced with a sequence of statements to save the current state, jump to the function to be executed and provide a location to return to. This stage comprises the following:

- A (normal) label is inserted directly after each function call with a unique name. In this implementation, the label name is constructed from the current function name with a unique number appended to it. The label marks the code which follows the function call and the point at which execution should resume once the call is complete.
- A statement is inserted before the function call to save the return address. A function call is made to store the address of the new label in a call activation record structure which is pushed onto the call stack. The address of the label is obtainable via a GCC extension (as per GCC manual [66], section 5.3). While this step requires a function call (as part of removing the function calls which exist in the input), marking this function as inline should result in stackless code.
- The function call itself must be replaced with a stackless function call. This is achieved through an ASM statement to jump to the label inserted at the top of the body of the function to be called. Obtaining the label identifier is simple since it is based directly on the name of the function.

Finally, it is necessary to transform return statements such that control is reverted to the caller function and the call stack is left in its previous state. The top element of the call stack is popped and the address of the label to return to is extracted from the call activation record. An inline assembly unconditional jump can revert control to this location. An example of this transformation is shown below in listing 5.10.

Listing 5.10: Function call post-data and call stackless transformation

```
// Following on from previous listing, there exists a transformed
    function, int foo(int a)
```

```
// This is transformed to the following:
void foo() {
        asm("foo_body_:");       // Inline label inserted
        ...
        asm("jmp
           %0"::"r"(pop_call_activation_record(global_stack)->return_addr));
           // Replace return statement with jump back to stored
           address of label
}

// The call to foo will be transformed to the following:
...
push_data_activation_record(global_stack,
   newDataActivationRecord(newFooParamStruct()));
((foo_param)global_stack->data_stack->top->data)->a = 4;

push_call_activation_record(global_stack,
   newCallActivationRecord(&&current_body_return_label));        //
   Wrap the address of the label to return to in a structure and push
   onto the call stack.
asm("jmp foo_body_;");          // Unconditional jump to foo
current_body_return_label:      // Add a label to return to

((current_body)global_stack->data_stack->top->prev->data)->i =
   ((foo_param)global_stack->data_stack->top->data)->__return_value;
   // Assignment of return value to i
pop_data_activation_record(global_stack);        // Remove foo
   structure from data stack
```

5.4 Other functionality

Outputting the C source code from its internal representation post-transformation is an important aspect of the program. This is achieved through a set of recursive functions which emit the appropriate tokens combined with the syntax which was removed when building the tree during the parsing phase. A function exists for each different type of token which has a unique set of operations associated with it (i.e. only one function exists for token types which denote a comma separated list) and a control function which determines which function should be executed. This could be developed into a framework which can handle string according to various templates, similar to functionality found within the ANTLR [48] project.

A set of auxiliary functions for internal processing of the data structures have been implemented. Any functionality related to tree manipulation may be better extracted and a framework for these tasks produced. Such a framework is discussed in section 6.4.

Chapter 6

Discussion

This chapter shall present the results of the project and analysis of these results. A critical evaluation of the implementation and the project as a whole follows, discussing the importance, impact and implications of different aspects of the project. The knowledge and experience gained is explained and original design choices are revisited. The final sections cover the future developments and potential of this project.

6.1 Results

The software for this project was developed and tested on an i686 running Fedora 8 [73] with GCC version 4.1. Freely available[1] software was used at all points of the project, since these tools were both effective, available and allows others to easily reproduce or continue this work.

Sample input and output source code (post-data stackless transformation and post-data + call stackless transformation) can be found in listings A.1, A.2 and A.3 in the appendix.

6.1.1 Analysis

The efficiency of the resultant code is reduced by three factors; heap allocation is considerably slower than stack allocation, accessing a variable requires multiple pointer traversals plus a cast, and the number of statements increases in almost every program. Increasing the efficiency is not particularly simple, not least because the ability to minimise stack usage requires trade-offs to be made. The traditional stack is used because it is both powerful and fast with architectural support. Heavy optimisation through inlining of functions and use of optimised inline assembly may help to improve the efficiency, however the latter increases the architecture specificity.

[1] Primarily “free as in speech”, and all “free as in beer”

6.2 Critical evaluation

The nature of this project is one of research and investigation; there are no hypotheses to be proved, nor detailed requirements against which this project are to be tested. Various techniques are considered, and some implemented, to discover what aspects of the project are possible and what methods are effective. Of equal importance are the less successful elements associated with this project - it is always important to learn and understand what does not work and why this so.

A proof of concept system has been implemented after consideration as to the best approach to use which is documented earlier in this report (section 4.6). The effectiveness of the system is a reasonable measure of how successful the proposed solution was and the existence of any unforeseen problems or delays. The transformations are complete enough to demonstrate a working proof of concept system, showing that the theory is sound. Sufficient thought and planning as to the applicability and usefulness of the transformations was given, however there are some limitations discussed later in the chapter relating to practical implementation issues.

6.2.1 Portability

To perform the control flow stackless transformation, it was necessary to use inline assembly code. Naturally this is architecture specific, however this does not render the tool or output of the tool completely unportable. Currently only unconditional jumps and labels are inserted into the code, therefore it should be trivial to replace these with the corresponding instructions for different architectures. If the registers are to be stored by the caller before each function call, considerably more assembly code will be required. This code could be taken from the implementation of ucontext or longjmp functionality. Similarly, this architecture specific code can be replaced depending on the system by simply substituting the appropriate sections of code for the desired architecture. While this document generally refers to the x86 architecture, this decision was made to simplify implementation and support. It should be noted that there is no use of any x86-specific techniques in the assembly used, which is why it is believed that substituting architecture specific assembly code should not be an issue.

6.2.2 Language choice

The choice of implementation language, C, was adequate. It seems only natural to manipulate C source in a program written in C, so as to keep considerations and cases which must be supported in mind whilst programming. A large number of tools and existing parsing and compiling based systems are written in C, a testament to the usefulness of the language to systems and lower level application programming. The use of Java or a similar object-oriented language (for which tools exist) could potentially be used, which may have some advantages if implementation details are changed considerably. In some

systems, particularly academic projects, functional programming languages are used for tree manipulation. Languages such as ML are particularly suited to this type of work due to the type inference and simplicity associated with building and modifying (tree-like) data structures. Using a functional programming language was considered in the early stages of the project, but the overhead associated with familiarisation of the language and subsequent debugging was seen to be too high.

6.2.3 Success criteria

This project had a number of objectives under the aim of creating a tool which would perform source-to-source translation. Therefore, the project can have varied degrees of success and accomplishment based on which of these aims are achieved. Success can be judged on the following criteria:

1. The presence of data stackless transformations. The effectiveness of the transformation and quality of the resultant source code will also be judged.
2. The presence of control flow stackless transformations. The effectiveness of the transformation and quality of the resultant source code will also be judged.
3. Any restrictions imposed on the input or output source.
4. Unsupported features of the language, caveats and similar issues. It should be noted whether these issues were as a result of a design decision, limitation of the algorithm(s) used or simply not achieved.
5. Code qualities - readability, portability of resultant source code.

6.2.4 Success against criteria

Data stackless transformations An effective data stackless transformation has been implemented, but there are edge cases which require further work. The transformation itself is sound in principle - structures can be generated and inserted into the source code for each environment. Loading and storing standard values works without issue, however processing parameters requires a little more work for some cases. Any function call which has function calls for or in its parameters can be comparatively hard to transform.

Control flow stackless transformations This transformation is not as complete or as effective as the data transformation. This is primarily due to the difficulty in correctly manipulating the tree. While conceptually simple, this transformation is hard to implement cleanly using just C. Future development of the project aims to fix the limitations and problems with this aspect of the system.

Restrictions Input source code is assumed to be correct to remove the requirement of a more advanced parsing phase designed to identify syntax and semantic errors. As previously mentioned, the C preprocessor is not supported (but an existing implementation could be introduced in the future), therefore it is necessary for the preprocessor to have run on the input source code, prior to being transformed.

Restrictions imposed Few restrictions are posed on the input source code, aside from it being valid and that it must not contain any preprocessor directives. It should be noted that the implementation does suffer from a number of cases where inputs fail to work. This is a target for improvement in the future.

Code quality The readability of the resultant code is likely to be considerably poorer than any reasonable input code. It should be noted that the readability of the code is not something which can be considered to be of utmost importance when using a transformation system to automatically produce a different version. After the data stackless transformation, function calls are still quite noticeable, but potentially tedious; after the control flow stackless transformation too, a minimum of five statements replace the original function call.

6.2.5 Limitations

There are some limitations and incomplete aspects of this implementation, however it is believed that the theory behind the system is sound and the problems are practical rather than conceptual. As discussed throughout this chapter, there are cases which were not anticipated, both at the conceptual and practical level; solutions are proposed for these issues.

The transformations are not guaranteed to work on all variables or function calls, as there are a number of complicated cases which require considerable processing. The complexity of the transformations therefore increases significantly when processing such cases, and is hindered by ensuring the consistency of the tree. An issue with regards to register values being overwritten due to the stackless calling mechanism was identified and a solution proposed. Similar issues are covered in later sections along with future developments.

Function calls as or in parameters

The structure to store the parameters for the initial function call being processed must be created and pushed onto the data stack. Each parameter field within the structure must be assigned a value. For expressions which do not contain a function call, it is relatively simple to match the actual parameter with the corresponding formal parameter and insert a statement to load the value into the structure. Any nested function calls must be extracted and executed before any functions which include the return address in a parameter can

execute. In this case, another structure must be created and pushed onto the stack and the process repeated recursively until all parameters for the "innermost" function call are loaded into the appropriate heap allocated structure. The innermost function can then be executed and the return value of this function can be substituted as the actual parameter for the next function call to execute. Once the return value has been loaded into the previous structure on the data stack, the structure for the innermost function (which has finished executing) can be popped.

This process is problematic because the structure on the top of the stack changes frequently if multiple nested function calls exist. The number of statements that must be inserted rises considerably with the number of nested function calls. New statements must be inserted in the correct location relative to the existing code and other new statements which have been added; this task however, is not sufficiently supported by the tree manipulation functions. This is one of the main motivators for a framework for modifying the parse tree as per the transformations.

6.3 Project continuation

6.3.1 Potential improvements

A number of improvements could be made which would complete and build on the immediate successes and achievements of this project. A number of developments to the application would help to transform it from a proof of concept to a more complete system, capable of handling all cases for C accurately and effectively. It is believed that a number of changes revolving around the alternative choices proposed in section 6.3.3 would be required for the system to be completed in this manner.

A list of potential improvements of varying size and complexity is listed below:

- Provide more customisation and options for the transformations. It may be beneficial to allow certain keywords to be overridden, or mark sections of code in the input to not be processed.
- Extend the parsing system to handle C preprocessor directives. The extent of support could vary from simply accepting include directives, to processing of text replacement.
- Support more architectures, or investigate an architecture agnostic approach to replace the inline assembly used for the control flow stackless transformations. See section 6.2.1 for further information about architecture support.
- Of considerable importance is the development of the control flow stackless transformations. The technique implemented is incomplete and must be developed further according to section 6.3.2. Code from ucontext, setjmp or even libc could be used as a basis.

- Refactor & restructure the code to facilitate further development.
- Replace the function calls inserted into the system with macros, inlined function calls or inline assembly code. This would enable the use of malloc and free to be changed easily.
- Insert C preprocessor macros into the resultant source code. These macros can be used to improve the readability of the resultant code.
- Provide a set of C preprocessor macros to enable the programmer to signal to the transformation system, or to partially implement stackless C without the use of transformations, comparable to Protothreads [38].
- Enable a visual representation of the tree. Modifying the current tree printing system to output code in the DOT format would allow graphs to be generated by Graphviz [74].
- Insert all structures generated as part of the data stackless transformation into an external file to increase cohesion by keeping all new stackless structures together.
- Create a new main function which initialises the heap-allocated stack and loads global variables, then calls original main function. This would simply separate the setup of the transformed program from the existing code.

6.3.2 Improving call stackless transformation

From preliminary testing, prior to the implementation of the transformations, a program transformed to be call stackless would work in some circumstances and not others. It is believed that the values stored in registers (and only registers) at the time of the unconditional jumps are lost. When control passes back to the caller function, values stored within registers are not in the expected state which can cause runtime errors in some cases. Since only some simple applications work without issue, the control flow stackless transformation is likely to be incomplete.

Two approaches could be used to rectify this issue. Within Extended ASM, it is possible to specify a list of registers which are "clobbered". A "clobbered" register is one which is used and modified by the provided assembly code, therefore any values loaded into these registers cannot be assumed to be valid by the compiler after the inline assembly has been executed. By signalling to the compiler that all registers are clobbered by the inserted code, the state of the registers will no longer be in an unexpected state after the code has been run. It is possible that by adding all registers to the clobber list when (unconditionally) jumping to a label (as part of a function call or return), the problem of runtime errors may be solved. Unfortunately this reduces the portability of the code since the registers will differ from architecture to architecture.

Alternatively, it may be possible to reuse code from the ucontext or longjmp family of functions. This functionality stores the current state of the application by dumping the

values of registers to a structure. The instruction pointer is included in the current state - this means that this functionality cannot be used directly within this system. Returning control from a called function to the scope of the caller by restoring the state saved by the caller would produce an infinite loop of function calls. By restoring the instruction pointer, the same set of instructions are repeatedly executed, hence why it is necessary to increment the instruction pointer. Much of the implementation of this functionality can be borrowed to store the state of all other registers into a structure which can be restored when control reverts to the caller scope, post-function call.

6.3.3 Alternative choices

As a result of some of the design decisions made earlier in the project, some parts of the implementation were more difficult. These decisions were made based on existing knowledge and considering how they would influence the implementation in terms of enabling or furthering development, compared with any additional complexity or time requirements. In almost all research and investigatory projects which do not include a big up-front design, there will be functionality which is both easier and harder to implement than expected.

Of particular note is the choice to implement a singly linked tree. With hindsight, making the tree doubly-linked such that there were two children nodes and an extra parent node would have simplified a number of tree manipulations. The original decision that a singly linked tree is simpler to implement and would suffice is still true, however the additional complexity in the data structure would help to reduce the complexity in the functions operating on the tree. A large number of the transformations operate at the statement level, but depend on the "contents" of these statements, i.e. expressions and sub-expressions. For example, a function call requires transformation, but may appear as an operand to another operator. The presence of a link to the parent node in the tree would allow a simple reverse/backwards traversal from the current location to find the start of the current statement.

A second improvement would be to perform further semantic analysis and transformation on the tree prior to processing. A list of statements which are represented as a linked list would be considerably easier to modify than the tree representation produced by the parser. The linked list would facilitate traversal of the list and insertion/deletion of nodes, which is otherwise complicated by the number of different cases which exist in a tree (see section 6.4).

Visualisation of the internal tree representation was neglected. With hindsight, it would have been wise to implement a method of visualising the tree at an early stage of the implementation to aid with debugging and highlight nodes which were affected by the transformations as they were applied.

6.3.4 Future developments

Future developments include functionality which was not strictly part of the initial project. The functionality and scope of this project is reasonably open-ended, therefore further investigation can focus on other related ideas, specialise the application towards some specific end, or even generalise the application to provide transformations for different cases.

Focusing on related ideas could involve the use of this system to achieve co-operative threading (as discussed in section 2.3.2); a list of pointers to the software-controlled, heap-allocated stacks could be maintained with one chosen to run as the active thread until it yields, then another stack can be switched to. Alternatively, attention could be shifted to improving tree rewriting languages for use in transformation systems and compilers. Specialising the application may involve enhancing the system to handle a specific subset (or superset of C), such as C written to some restrictive coding standards or C which leverages the preprocessor or GCC extensions [66]. Generalisation could be achieved by focusing on a tree manipulation framework, a more general transformation engine, or the unification of both to provide a system which can manipulate source code represented as a parse tree according to a set of input transformations.

Since the software implemented for this project is a proof of concept and the feasibility of the system has been demonstrated, the software could be rewritten according to a more developed set of requirements.

6.4 Tree manipulation framework

Conceptually, a number of steps in the process of applying the data and call stackless transformations are simple. Practically, these are quite complex to successfully implement because a considerable amount of effort is required to handle edge cases. Such steps can be the result of shortcomings in a design, or in the chosen programming language; however, it is believed that neither is the case and a mis-estimation of the task is responsible.

The main tasks which are deemed to be conceptually simple, but practically problematic centre around the manipulation of the tree which represents the source code. Adding a statement into the tree is an example of such a task because the request is poorly defined (i.e. not explicit) and a number of nodes are involved in the completion of this task. The request to add a statement is poorly defined because it is not clear where the statement is to be added, e.g. a statement may be inserted before or after a specified statement, at the beginning or end of a list, before or after structure declarations/function definitions, or in a number of alternative methods depending on the transformation being performed.

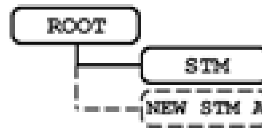

Figure 6.1: Inserting a statement into tree

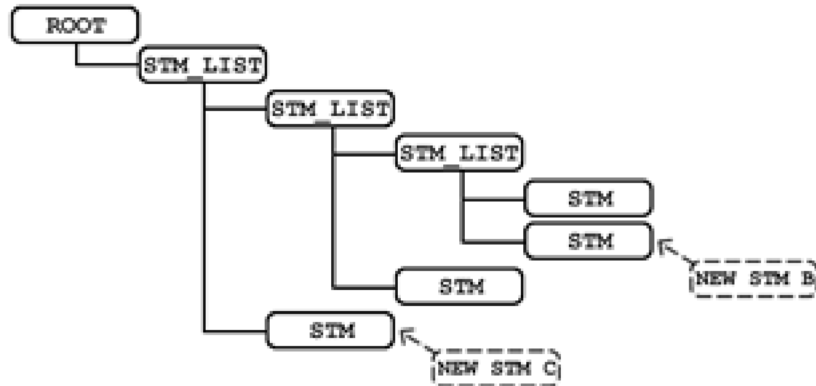

Figure 6.2: Inserting a statement into tree

In the case of a statement being inserted directly after a specified statement, one of the following cases may arise:

- There is only one statement in the current scope, therefore there is no statement list node in the tree. A statement list node must be added, with children of the existing statement and the new statement. This is shown in figure 6.1 with statement A being inserted.
- There are multiple statements in the current scope and the new statement is to be inserted after the first statement. This requires the end of the statement list to be found and a new statement list node added at the end to accommodate the new statement. This is shown in figure 6.2 with statement B being inserted.
- There are multiple statements in the current scope and the new statement is to be inserted after any other statement. The parent of the "previous" statement must be located and a new statement list node added at this point to accommodate the new statement. This is shown in figure 6.2 with statement C being inserted.

This list demonstrates the complexity of adding a statement to the list and this functionality may be best built into a framework which provides an API to tree manipulation functionality. A framework which provides the necessary functionality for manipulating the tree would be of great benefit to the project. As previously discussed, the steps which pose implementation problems are related to the tree - by abstracting the complexity of

manipulating the tree, it is easier to focus on the transformation being applied. This abstraction will help to separate the code transformations from the tree manipulations, or even hide the use of the tree entirely.

It may be useful to (formally) define the structure of the parse tree, to enable the development of a specialised framework operating on a tree which conformed to a particular standard.

6.5 Summary

This chapter has discussed the results obtainable from the implementation of the data and call stackless transformations and considered the effects on the efficiency of the code and similar issues which dictate the success of this implementation, and in a broader scope, the project as a whole.

The implementation relies on compiler frontend knowledge and technology to recognise the input C code. It is believed that this phase works without issue. A parse tree is produced and semantic analysis performed resulting in tree annotations and internal structures modelling aspects of the input code. This phase is also effective and complete. Using the techniques chosen as part of the potential solution, further analysis and transformation of the tree takes place. The implemented transformations are not as powerful as anticipated due to practical issues regarding tree manipulation. Finally, the resultant tree is output to produce C source code once more. Some elements of the project were significantly harder to implement or solve than others, e.g. care must be taken not to introduce any cycles into the tree as modifications are applied.

A number of improvements and changes to the system are proposed. It has been shown that transformations are a feasible and effective method of achieving the aims of the project, and there are still further problems to be solved in this area. Of particular note is the idea of a tree manipulation framework and the benefits which could be gained from this. In the case of ensuring the stability of call stackless transformed code, further investigation will be required into storing the values of the registers before jumping to the called function.

Further development has been identified around the project in general, by continuing the development of this project along similar lines, or by specialising or generalising the application of the fields covered within this project. With more time, the system could have been brought from a proof of concept implementation to a fully fledged system, capable of identifying and handling all edge cases, as well as being built on a modular system which would facilitate reuse for similar transformation or tree manipulation based systems.

Chapter 7

Conclusion

In terms of meeting the aims of the project outlined in section 1.1, this project has been a success. An implementation of stackless C has been produced via a source to source translator, reading in a C source file and outputting code which minimises the use of the program stack. This is achieved by transferring all stack allocated data to the heap.

With regards to the objectives of the project outlined in section 1.2, this project has also been a success. Drawing from literature which motivates the project primarily in terms of security, overcoming limits imposed by the use of a stack and debugging, consideration has been given to existing software and findings in related fields. From knowledge in these fields, combined with research into existing tools and the details surrounding the notion of stackless C, a range of techniques were formulated and evaluated. Designing the transformations followed on from selecting the most appropriate and feasible techniques in the form of a proposed solution.

The method behind the transformations, specifically how these might be implemented via a source to source rewriting system operating on a tree built up from the input file was presented. A proof of concept application was produced from the design and method which implemented the set of transformations on a limited section of valid C. Evaluation of the implementation, noting successful & problematic aspects, and considering future work and contribution of this project fulfils the final objective established.

The result of the project is the proof of concept application, and the knowledge derived from both existing systems and this system in terms of design, implementation and evaluation; this information is conveyed within this report. The implementation itself can be considered to be a partial success. A successful parsing and semantic analysis phase is provided and many of the transformations implemented work without issue. The resultant tree is transformed back into C source code. There are considerable amounts of valid C which are not supported by this tool, and not all transformations worked as planned. Implementation progress was slower than expected, compounded by difficulties in debugging the code due to the lack of a visual representation of the large parse tree being operated on. A number of lessons have been learnt from this and are discussed in the project discussion (chapter 6).

With regards to the idea of being *stackless*, the project can be considered a success. The proposed solution produces code which is stackless, but has highlighted some of the more difficult aspects of the project. Two main problems arose from the issue of stacklessness: the difficulty in handling function calls nested within other function calls as part of the data stackless transformation, and the necessity of loading and storing register values before jumping to another point within the application.

Evaluating the project as a whole, the author believes it can be classed a success based on the achievements described, the knowledge gained and the lessons learnt throughout. The practical contributions include an investigation into the techniques and important aspects of the C programming language with regards to implementing a "stackless C". Despite aspects of the system being less advanced or robust as desired, the theory and ideas driving the project have been shown to be sound on this scale. The importance of handling and modifying a parse tree has been noted, and it is believed that similar source to source systems would benefit greatly from a support framework which provides tree manipulation functionality.

Future work has been proposed in several areas - developing the stackless C system further, to produce a robust and complete implementation; investigating alternative techniques to produce stackless C or any similar transformation; and generalisation of source to source/tree rewriting systems to perform modifications according to a specification provided.

Chapter 8

Bibliography

A note on literature It should be noted that a reasonable number of new techniques in the field of security, especially with regards to vulnerabilities crop up in hacker and cracker publications before academic-style literature or mainstream books[22]. This material is still of interest despite its origin, not least because the authors may be experts in their field and their writings read and reviewed by many, albeit not in the same manner or rigour associated with an academic journal publication.

[1] Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, December 1997.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[3] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *In USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[4] Dennis M. Ritchie. The development of the C language. *ACM SIGPLAN Notices*, 28(3):201–208, 1993.

[5] OED Online, editor. *Oxford English Dictionary.* Oxford University Press, 2nd edition, 1989.

[6] Wikipedia. Call stack — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 22-November-2007].

[7] Philip Koopman. *Stack Computers – The New Wave.* Ellis Horwood, 1989.

[8] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture.* Intel Corporation, 1 edition, 11 2007.

[9] BugHunter. Articles and security papers about buffer overflows. `http://doc.bughunter.net/buffer-overflow/`. [Online; accessed 22-November-2007].

[10] H. Prokop. Cache-oblivious algorithms, June 1999.

[11] Wikipedia. Processor register — Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 2-December-2007].

[12] Denis Howe. Free Online Dictionary of Computing. `http://foldoc.org/`.

[13] Henry J. Bowlden. An introduction to ALGOL 68. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, volume 28, pages 345–346, New York, NY, USA, March 1993. ACM Press.

[14] Donald E. Knuth. The remaining trouble spots in ALGOL 60. *Commun. ACM*, 10(10):611–618, October 1967.

[15] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[16] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Network and Distributed System Security Symposium(NDSS)*, pages 149–162, San Diego, California, February 2003.

[17] Guido Hogen and Rita Loogen. A new stack technique for the management of runtime structures in distributed implementations. Technical Report 93-3, Technical Report 93-03, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Germany, 1993.

[18] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[19] James S. Miller and Guillermo J. Rozas. Garbage collection is fast, but a stack is faster. Technical Report AIM-1462, Cambridge, MA, USA, 1994.

[20] Nikolaos S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards & Interfaces*, 23(3):169–185, July 2001.

[21] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[22] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *Security & Privacy Magazine, IEEE*, 2(4):20–27, 2004.

[23] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 227–237, 2003.

[24] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[25] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[26] H. Etoh. GCC extension for protecting applications from stack-smashing attacks (ProPolice). `http://www.trl.ibm.com/projects/security/ssp/`, 2003. `http://www.trl.ibm.com/projects/security/ssp/`.

[27] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *In 10th USENIX Security Symposium, Washington, D.C*, pages 55–66, August 2001.

[28] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, and A. Jalote. SmashGuard: A hardware solution to prevent security attacks on the function return address. *Computers, IEEE Transactions on*, 55(10):1271–1285, 2006.

[29] J. Mcgregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. In *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, pages 243–250, 2003.

[30] Andreas Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[31] Jeremy Condit, Matthew Harren, Scott Mcpeak, George C. Necula, and Westley Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, volume 38, pages 232–244. ACM Press, May 2003.

[32] George C. Necula, Scott Mcpeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 37, pages 128–139. ACM Press, January 2002.

[33] George C. Necula, Jeremy Condit, Matthew Harren, Scott Mcpeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005.

[34] George C. Necula, Jeremy Condit, Matthew Harren, Scott Mcpeak, and Westley Weimer. CCured documentation. `http://manju.cs.berkeley.edu/ccured/`, January 2007. [Online; accessed 22-November-2007].

[35] Dan Grossman. Type-safe multithreading in cyclone. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 13–25, New York, NY, USA, 2003. ACM Press.

[36] Frdric Peschanski and Samuel Hym. A stackless runtime environment for a pi-calculus. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 57–67, New York, NY, USA, 2006. ACM Press.

[37] Christian Tismer. Continuations and Stackless Python. Technical report, In Proceedings of the 8th International Python Conference, 2000.

[38] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[39] FSF. *GNU Marst Readme*.

[40] Manish Prasad and Tzi C. Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference*, 2003.

[41] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, volume 30, pages 291–300, New York, NY, USA, June 1995. ACM Press.

[42] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM.

[43] Brian Bershad, Dennis Lee, Ted Romer, Geoff Voelker, Alec Wolman, Wayne Wong, Brad Chen, and Hank Levy. Etch overview. `http://etch.cs.washington.edu/overview/index.html`, 1997.

[44] Lu Xun. A Linux executable editing library (LEEL). Master's thesis, National University of Singapore, 1999.

[45] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 33(3):60–66, March 2000.

[46] E. Gagnon and L. Hendren. Sablecc – an object-oriented compiler framework, 1998.

[47] Sreeni Viswanadha, Sriram Sankar and contributors. Java Compiler Compiler (JavaCC) - The Java Parser Generator. `https://javacc.dev.java.net/`. [Online; accessed 16-April-2008].

[48] Terence J. Parr and Russell W. Quong. Antlr: A predicated-LL(k) parser generator. *Software - Practice and Experience*, 25(7):789–810, 1995.

[49] Scott McPeak and George C. Necula. Elkhound: A fast, practical glr parser generator. In *CC*, pages 73–88, 2004.

[50] Todd Austin, Scott Breach, and Guri Sohi. SCC: The safe C compiler. `http://pages.cs.wisc.edu/~austin/scc.html`, September 1995.

[51] Devin Cook. GOLD Parsing System - A Free, Multi-Programming Language, Parser. `http://www.devincook.com/goldparser/`, 2007. [Online; accessed 16-April-2008].

[52] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer, Steven W. K. Tjiang, Shih W. Liao, Chau W. Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[53] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom. Flick: A flexible, optimizing IDL compiler. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–56, 1997.

[54] Hipp, Wyrick & Company, Inc. (Hwaci). The LEMON Parser Generator. `http://www.hwaci.com/sw/lemon/`. [Online; accessed 16-April-2008].

[55] Hanspeter Mssenbck, Markus Lberbauer, Albrecht W. The Compiler Generator Coco/R. `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/`. [Online; accessed 16-April-2008].

[56] Lars M. Garshol. BNF and EBNF: What are they and how do they work? `http://www.garshol.priv.no/download/text/bnf.html`, July 2003. [Online; accessed 16-April-2008].

[57] Free Software Foundation. Bison - GNU parser generator. `http://www.gnu.org/software/bison/`, April 2007. [Online; accessed 16-April-2008].

[58] Free Software Foundation. Flex - a scanner generator [manual]. http://www.gnu.org/software/flex/manual/, November 1998. [Online; accessed 16-April-2008].

[59] Jeff Lee, Jutta Degener. ANSI C Yacc grammar. http://www.quut.com/c/ANSI-C-grammar-y.html, December 2006. [Online; accessed 16-April-2008].

[60] Jeff Lee, Jutta Degener. ANSI C grammar, Lex specification. http://www.quut.com/c/ANSI-C-grammar-l-1998.html, December 2006. [Online; accessed 16-April-2008].

[61] Program-transformation.org contributors. Program Transformation - Program-Transformation.Org: The Program Transformation Wiki. http://www.program-transformation.org/Transform/ProgramTransformation, April 2007. [Online; accessed 16-April-2008].

[62] Dennis M. Ritchie. *C Reference Manual*. Bell Telephone Laboratories, Murray Hill, New Jersey 07974.

[63] Tony Zhang. *Sams Teach Yourself C in 24 Hours*. Macmillan Computer Publishing, first edition, 1997.

[64] Derek Jones. *The New C Standard: An Economic and Cultural Commentary*. Cisco Press, 1.1 edition, January 2008.

[65] Free Software Foundation. The C Preprocessor [manual]. http://gcc.gnu.org/onlinedocs/cpp/index.html, 2007. [Online; accessed 16-April-2008].

[66] Free Software Foundation. Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc/index.html, 2008. [Online; accessed 16-April-2008].

[67] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone.

[68] Sandeep S. GCC-Inline-Assembly-HOWTO. http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html, March 2003. [Online; accessed 16-April-2008].

[69] Phillip. Using Assembly Language in Linux. http://asm.sourceforge.net/articles/linasm.html, January 2001. [Online; accessed 16-April-2008].

[70] Brennan Underwood. Brennan's Guide to Inline Assembly. http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html, 1996. [Online; accessed 16-April-2008].

[71] Bharata Rao. Inline assembly for x86 in Linux. http://www-128.ibm.com/developerworks/library/l-ia.html, March 2001. [Online; accessed 16-April-2008].

[72] E. Dijkstra. Go to statement considered harmful. pages 27–33, 1979.

[73] Red Hat Inc. and others. Fedora Project. http://fedoraproject.org/, 2008. [Online; accessed 16-April-2008].

[74] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

Appendix A

Code

A.1 Results

A.1.1 Input code

Listing A.1: Input test code

```
#include <stdlib.h>
#include <stdio.h>

static int an_int;

typedef struct variable_ {
    int                 types;
    unsigned int                string_id;
    char                        *string;
    struct variable_    *next;
} *Variable;

struct nv_params {
    int types;
    unsigned int string_id;
    char *s;
};

float f = 93.4;

Variable newVariable(int types, unsigned int string_id, char *s) {
    unsigned int i = 9, j, k;
    Variable v = malloc(sizeof(*v));
    v->types = types;
    v->string_id = string_id;
    v->string = s;
    v->next = NULL;

    k = 10;

    return v;
}

int global_int = 4;
```

```
char global_char;

void whilefn(Variable v) {
     while(++global_int < 10) {
          printf("%d\n", v->string);
     }

     switch(v->string_id) {
     default:
              printf("default\n");
              break;
     case 4:
          return;
     }
     return;
}

int loop() {
    int i, j = 0;
    for (i = 0; i < 10; i++) {
        j *= 2;
    }
    return j;
}

int main() {
    double d = 7.8;
    newVariable(-12, 4, "bob");
    global_int++;
    return 1;
}
```

A.1.2 Post data stackless transformation

Listing A.2: Data stackless transformed code

```
#include <stdlib.h>
#include <stdio.h>
#include "actrecord.h"

typedef struct dummy__ *dummy;

static GlobalStack  gs;

typedef struct variable_ {
  int types;
  unsigned int string_id;
  char *string;
  struct variable_*next;
} *Variable;

struct nv_params {
  int types;
  unsigned int string_id;
  char *s;
};

typedef struct global6000__ {
  int an_int;
  float f;
  int global_int;
```

```
  char global_char;
} *global6000;

typedef struct newVariable6001__ {
  int types;
  unsigned int string_id;
  char *s;
  Variable rv__;
} *newVariable6001;

typedef struct newVariable6002__ {
  unsigned int k;
  unsigned int j;
  unsigned int i;
  Variable v;
} *newVariable6002;

typedef struct whilefn6003__ {
  Variable v;
} *whilefn6003;

typedef struct loop6007__ {
  int rv__;
} *loop6007;

typedef struct loop6008__ {
  int j;
  int i;
} *loop6008;

typedef struct main6011__ {
  double d;
} *main6011;

void newVariable() {
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(newVariable6002))));
  ((newVariable6002)gs->data_stack->data)->i = 9;
  ((newVariable6002)gs->data_stack->data)->v =
      malloc(sizeof(*((newVariable6002)gs->data_stack->data)->v));
  ((newVariable6002)gs->data_stack->data)->v->types =
      ((newVariable6001)gs->data_stack->prev->data)->types;
  ((newVariable6002)gs->data_stack->data)->v->string_id =
      ((newVariable6001)gs->data_stack->prev->data)->string_id;
  ((newVariable6002)gs->data_stack->data)->v->string =
      ((newVariable6001)gs->data_stack->prev->data)->s;
  ((newVariable6002)gs->data_stack->data)->v->next = NULL;
  ((newVariable6002)gs->data_stack->data)->k = 10;
  ((newVariable6001)gs->data_stack->prev->data)->rv__ =
      ((newVariable6002)gs->data_stack->data)->v;
  pop_data_act_record(gs);
}

void whilefn() {
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(dummy))));
  while (++((global6000)gs->data_stack->prev->data)->global_int < 10) {
    printf("%d\n", ((whilefn6003)gs->data_stack->data)->v->string);
  }
  switch (((whilefn6003)gs->data_stack->data)->v->string_id) {
    default:
    printf("default\n");
    break;
    case 4:
```

```
    pop_data_act_record(gs);
    return;
  }

  pop_data_act_record(gs);
  return;
}

void loop() {
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(loop6008))));
  ((loop6008)gs->data_stack->data)->j = 0;
  for (((loop6008)gs->data_stack->data)->i = 0; ((loop6008)gs->data_stack->data)->i
      < 10; ((loop6008)gs->data_stack->data)->i++) {
    ((loop6008)gs->data_stack->prev->data)->j *= 2;
  }
  ((loop6007)gs->data_stack->prev->data)->rv__ =
      ((loop6008)gs->data_stack->data)->j;
  pop_data_act_record(gs);
}

int main() {
  gs = newGlobalStack();
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(global6000))));
  ((global6000)gs->data_stack->data)->f = 93.4;
  ((global6000)gs->data_stack->data)->global_int = 4;
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(main6011))));
  ((main6011)gs->data_stack->data)->d = 7.8;
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(newVariable6001))));
  ((newVariable6001)gs->data_stack->data)->types = -12;
  ((newVariable6001)gs->data_stack->data)->string_id = 4;
  ((newVariable6001)gs->data_stack->data)->s = "bob";
  newVariable();
  printf("global int: %d, d: %g, variable string: %s\n",
      ((global6000)gs->data_stack->prev->prev->data)->global_int,
      ((main6011)gs->data_stack->prev->data)->d,
      ((newVariable6001)gs->data_stack->data)->s);
  pop_data_act_record(gs);
  ((global6000)gs->data_stack->prev->data)->global_int++;
  pop_data_act_record(gs);
  return 1;
}
```

A.1.3 Post data + call stackless transformation

Listing A.3: Data + call stackless transformed code

```
#include <stdlib.h>
#include <stdio.h>
#include "actrecord.h"

typedef struct dummy__ *dummy;

static GlobalStack  gs;

typedef struct variable_ {
  int types;
  unsigned int string_id;
  char *string;
  struct variable_*next;
} *Variable;
```

```
struct nv_params {
  int types;
  unsigned int string_id;
  char *s;
};

typedef struct global6000__ {
  int an_int;
  float f;
  int global_int;
  char global_char;
} *global6000;

typedef struct newVariable6001__ {
  int types;
  unsigned int string_id;
  char *s;
  Variable rv__;
} *newVariable6001;

typedef struct newVariable6002__ {
  unsigned int k;
  unsigned int j;
  unsigned int i;
  Variable v;
} *newVariable6002;

typedef struct whilefn6003__ {
  Variable v;
} *whilefn6003;

typedef struct loop6004__ {
  int rv__;
} *loop6004;

typedef struct loop6005__ {
  int j;
  int i;
} *loop6005;

typedef struct foo6006__ {
  int a;
  int rv__;
} *foo6006;

typedef struct main6007__ {
  double d;
  Variable v;
} *main6007;

void newVariable() {
  asm("newVariable__:");
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(newVariable6002))));
  ((newVariable6002)gs->data_stack->data)->i = 9;
  ((newVariable6002)gs->data_stack->data)->v =
      malloc(sizeof(*((newVariable6002)gs->data_stack->data)->v));
  ((newVariable6002)gs->data_stack->data)->v->types =
      ((newVariable6001)gs->data_stack->prev->data)->types;
  ((newVariable6002)gs->data_stack->data)->v->string_id =
      ((newVariable6001)gs->data_stack->prev->data)->string_id;
  ((newVariable6002)gs->data_stack->data)->v->string =
      ((newVariable6001)gs->data_stack->prev->data)->s;
```

```
  ((newVariable6002)gs->data_stack->data)->v->next = NULL;
  end:
  ((newVariable6002)gs->data_stack->data)->k = 10;
  ((newVariable6001)gs->data_stack->prev->data)->rv__ =
      ((newVariable6002)gs->data_stack->data)->v;
  pop_data_act_record(gs);
  asm("jmp %0"::"r"(pop_call_act_record(gs)->ra__));
}

/*void whilefn() {
  asm("whilefn:");
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(dummy))));
  bar();
  top:
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(foo6006))));
  push_call_act_record(gs, newCallActRecord(&&whilefn2006));
  asm("jmp foo");
  whilefn2006:
  pop_data_act_record(gs);
  while (++((global6000)gs->data_stack->prev->prev->data)->global_int < 10) {
    printf("%d\n", v->string);
  }
  switch (((whilefn6003)gs->data_stack->prev->data)->v->string_id) {
    default:
    printf(""default\n"");
    break;
    case 4:
    goto top;
    pop_data_act_record(gs);
    asm("jmp %0"::"r"(pop_call_act_record(gs)->ra__));
  }

  pop_data_act_record(gs);
  asm("jmp %0"::"r"(pop_call_act_record(gs)->ra__));
  }*/

void loop() {
  asm("loop__:");
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(loop6005))));
  ((loop6005)gs->data_stack->data)->j = 0;
  for (((loop6005)gs->data_stack->data)->i = 0; ((loop6005)gs->data_stack->data)->i
      < 10; ((loop6005)gs->data_stack->data)->i++) {
    ((loop6005)gs->data_stack->data)->j *= 2;
  }
  ((loop6004)gs->data_stack->prev->data)->rv__ =
      ((loop6005)gs->data_stack->data)->j;
  pop_data_act_record(gs);
  // missing return statement - therefore no stackless return!
}

void foo() {
  asm("foo__:");
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(dummy))));
  ((foo6006)gs->data_stack->prev->data)->a + 2;
  ((foo6006)gs->data_stack->prev->data)->rv__ =
      ((foo6006)gs->data_stack->prev->data)->a * 2;
  pop_data_act_record(gs);
  // missing return statement - therefore no stackless return!
}

int main() {
  asm("main__:");
```

```
  gs = newGlobalStack();
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(global6000))));
  ((global6000)gs->data_stack->data)->f = 93.4;
  ((global6000)gs->data_stack->data)->global_int = 4;
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(main6007))));
  ((main6007)gs->data_stack->data)->d = 7.8;
  push_data_act_record(gs, newDataActRecord(malloc(sizeof(newVariable6001))));
  ((newVariable6001)gs->data_stack->data)->types = -12;
  ((newVariable6001)gs->data_stack->data)->string_id = 4;
  ((newVariable6001)gs->data_stack->data)->s = "bob";
  push_call_act_record(gs, newCallActRecord(&&main2020));
  asm("jmp newVariable__");
  main2020:
  ((main6007)gs->data_stack->prev->data)->v =
      ((newVariable6001)gs->data_stack->data)->rv__;
  pop_data_act_record(gs);
  ((global6000)gs->data_stack->prev->data)->global_int++;
  printf("global int: %d, var->string_id = %u, var->string = %s\n",
      ((global6000)gs->data_stack->prev->data)->global_int,
      ((newVariable6001)gs->data_stack->data)->string_id,
      ((newVariable6001)gs->data_stack->data)->string_id);
  pop_data_act_record(gs);
  return 1;
}
```

A.2 Source code

A.2.1 Compilation & execution notes

- This system has been developed using GCC.
- The Boehm-Demers-Weiser garbage collector is used and therefore must be installed.
- A Makefile is included with the program source code; make will suffice to compile the code.
- The name of the compiled application is *c2c*.
- An input file is read from standard input, and output is via standard output. "./c2c < input.c > output.c" would therefore be a standard method of using the application.

A.2.2 Source code excerpts

Listing A.4: Source code excerpts from main transformation (c2c.c)

```
/**
 * STRUCT BUILDER
 */

/**
 * Calls tree walker to walk tree, generating structs for appropriate nodes.
 */
void build_structs_for_environments(Node tree) {
  tree_walker(tree, &struct_generator);
}
```

```
/**
 * Generates structs for nodes which have environments.
 */
void struct_generator(Node t) {
  static Node root = NULL;
  Node struct_def_node;

  // Store location of root
  if (root == NULL) {
    root = t;
  }

  // Find environments & generate structs
  if (t->env) {
    struct_def_node = build_struct_for_environment(t->env);

    if (struct_def_node != NULL ) {
      if (debug_mode()) {
        print_tree(struct_def_node);
        emitter(struct_def_node);
                printf("\n\n\n");
      } else {
        printf("Struct generated\n");
      }

      insert_struct_into_tree(root, struct_def_node);
    } else {
      printf("Struct not generated\n");
    }
  }
}

/**
 * Build and return structure corresponding to specified environment
 */
Node build_struct_for_environment(Environment e) {
  static unsigned int unique_environment_id = 6000;
  FnEntry current_fne;
  string struct_name, typedef_name;
  char tmp_buf[BUFFER_LEN];
  Node struct_decl = NULL;
  Node struct_node = NULL;

  assert(e != NULL);

  memset(tmp_buf,'\0',BUFFER_LEN); // Initialise buffer

  // If environment has no variables, do not need to create a struct!
  if((e->varlist != NULL && e->varlist->head != NULL) || (e->type == FNDEF &&
      get_current_fne(e)->string_id != get_string_id(st, MainFunctionIdentifier))) {
    // Find the parent environment with an attached function entry
    current_fne = get_current_fne(e);

    // Generate the unique struct name for this environment (containing the fnname)
    if (current_fne) {
      struct_name = String(current_fne->fnname); // make a copy of the fnname
    } else {
      struct_name = String("global");
    }

    // Combine fnname + id to give structure name
```

```
    strncpy(tmp_buf, struct_name, BUFFER_LEN-6);
    strncat(tmp_buf, itoa(unique_environment_id++), 4);
    typedef_name = String(tmp_buf);
    strncat(tmp_buf, "__", 2);
    struct_name = String(tmp_buf);

    // Update env to include structname
    e->structname = typedef_name; //struct_name;

    // Build the contents of the structure if the environment has variables declared
    if (e->varlist != NULL && e->varlist->head != NULL) {
      struct_decl = build_struct_decl(e->varlist);
    }

    // If the structure is for a fn definition, add the return value field
    if (e->type == FNDEF && e->fne) {
      // Don't add a return field for the main fn
      if (e->fne->string_id != get_string_id(st, MainFunctionIdentifier)) {
            e->has_return_field = true;
        struct_decl = add_return_value_to_struct(struct_decl, e->fne);
      }
    }

    if (struct_decl != NULL) {
      // Create the typedef'd struct
      struct_node = build_typedef_struct_node(struct_name, struct_decl,
          typedef_name);
    }
    return struct_node;
  }

  // No struct to be generated due to empty environment
  // Return nothing
  printf("Environment is empty!\n");
  return NULL;
}

/**
 * Add return value field for specified function to a structure declaration.
 */
Node add_return_value_to_struct(Node struct_decl, FnEntry fne) {
  Node return_val_decl;

  // Determine if return type of function is void
  if (is_typelist_void(fne->returntypes)) {
    // Void return type. do nothing
    return struct_decl;
  } else {
    return_val_decl = newNode(STRUCT_DECL,
                              build_types_for_typelist(fne->returntypes),
                              newLeaf(IDENTIFIER,
                                  newTokenId(StructureReturnValueFieldIdentifier)));

    // If there is no existing structure, dont need to add a list
    if (struct_decl == NULL) {
      return return_val_decl;
    } else {
      return newNode(STRUCT_DECL_LIST,
                     struct_decl,
                     return_val_decl);
    }
  }
```

```
}

/**
 * Adds typedef and structure wrapper to a list of fields
 */
Node build_typedef_struct_node(string struct_name, Node struct_decl, string
    typedef_name) {
  Node struct_node = newNode(DECL,
                             newNode(DECL_SPEC_STOR,
                                     newLeaf(TYPEDEF, NULL),
                                     newNode(DECL_SPEC_TYPE,
                                             newNode(STRUCT,
                                                     newLeafId(struct_name),
                                                     struct_decl),
                                             NULL)),
                             newNode(INIT,
                                     newLeafId(typedef_name),
                                     NULL));

  add_var_pointer(struct_node); // make it a pointer to a type
  set_transform_lock_node(struct_node); // make sure the struct is not processed
      further.
  return struct_node;
}

/**
 * Recursively build and return the fields of a structure corresponding to a list
     of variables
 */
Node build_struct_decl(VariableList el) {
  Node sub_struct_decl;

  if (el->tail == NULL) {
    dprintf("build_struct_decl. no tail\n");
    sub_struct_decl = build_struct_for_var(el->head);
  } else if (el->tail != NULL && el->tail->tail == NULL) {
    dprintf("build_struct_decl. tail then no tail\n");
    sub_struct_decl = newNode(STRUCT_DECL_LIST,
                              build_struct_for_var(el->head),
                              build_struct_for_var(el->tail->head));
  } else {
    dprintf("build_struct_decl. otherwise\n");
    sub_struct_decl = newNode(STRUCT_DECL_LIST,
                              build_struct_for_var(el->head),
                              build_struct_decl(el->tail));
  }
  return sub_struct_decl;
}

/**
 * ENVIRONMENT BUILDER
 */

/**
 * Build environments for entire tree
 */
void build_environments_for_tree(Node tree) {
  environment_builder(st, tree->env, tree);
}

/**
```

```
 * Recursively build environments for entire tree. Current node is 't', current
     environment is 'e'.
 */
void environment_builder(StringTable st, Environment e, Node t) {
  static bool new_fn_body = false;
  bool new_env = false;
  unsigned int string_id;
  string varname;
  TypeList typelist;
  VariableList var_list;
  FnEntry fne;

  // Set fnbody flag
  if (t->type == FNBODY) {
    new_fn_body = true;
  }

  // Create new environments for fn definitions (parameters) and any compound
      statements
  if (t->type == FNDEF || t->type == CMPD) {
    new_env = true;
    dprintf("Create new environment\n");
    if (new_fn_body) {
      e = newEnvironment(e, FNBODY);
      new_fn_body = false;
    } else {
      e = newEnvironment(e, t->type);
    }
    t->env = e; // Attach this new environment to the current Node.

    if(t->type == FNDEF) {
      // Add the fn def to the _parent_ environment
      fne = build_fnentry_for_fndef(st, t);
      add_fnentry_to_env(e->parent, fne);
      set_fnentry_for_env(e, fne);
      printf("Add fn to environment\n");
    } else {
      set_fnentry_for_env(e, e->fne);
    }
  }

  // Process variables & parameters
  if(t->type == DECL) {
    if (t->left && t->left->type == DECL_SPEC_STOR && t->left->left &&
        t->left->left->type == LEAF && t->left->left->subtype == TYPEDEF) {
      // Typedef, so skip
    } else if (t->right) {
      typelist = get_type_list_for_decl(st, t->left);
      var_list = get_var_names(st, typelist, t->right, NULL);
      add_envlist_to_env(e, var_list);
    } // else {
    // probably a struct definition

  } else if(t->type == PARAM) {
    typelist = get_type_list_for_param(st, t->left);
    varname = get_var_name(t->right);
    string_id = lookup_string(st, varname);
    printf("Add %s to parameter environment\n", varname);
    add_variable_to_env(e, newVariable(typelist, string_id, varname, t,
        t->right->type));
  } else {
    if (t->left)  environment_builder(st, e, t->left);  // Recurse left
```

```
    if (t->right) environment_builder(st, e, t->right); // Recurse right
  }

  // Restore the previous environment
  if (new_env) {
    e = e->parent;
  }
}

/**
 * Recursively build and return a variable list of actual parameters
 */
VariableList get_actual_params(StringTable st, Node t, VariableList var_list) {
  string varname;
  unsigned int string_id;
  TypeList tl;

  if (t == NULL) {
    return NULL;
  } else if (t->type == PARAM_LIST) {
    var_list = get_actual_params(st, t->left, var_list);
    return get_actual_params(st, t->right, var_list);
  } else if(t->type == PARAM) {
    tl = get_type_list_for_param(st, t->left);
    varname = get_var_name(t->right);
    string_id = lookup_string(st, varname);
    printf("Add %s to varlist\n", varname);
    return newVariableList(newVariable(tl, string_id, varname, t, t->right->type),
        var_list);
  } else {
    perror("Shouldn't be here");
  }
}

/**
 * TREE BUILDING FUNCTIONS
 */

/**
 * Heap allocate a struct for an environment
 */
Node get_heap_alloc_struct_for_env_tree(Environment e) {
  Node return_node;
  string structname;

  // If there is no struct name, we must make a dummy struct to simplify push and
      pop from the stack
  if (e->structname) {
    structname = e->structname;
  } else {
    structname = DataActRecordDummyTypedefIdentifier;
  }

  return_node = newNode(FN_CALL,
                        newLeafId(MallocFnIdentifier),
                        newNode(SIZEOF_TYPE,
                                newLeafId(structname),
                                NULL));
  set_transform_lock_node(return_node);
  return return_node;
}
```

```
/**
 * Create tree for specified field name in particular scope
 */
Node get_data_actrecord_scope_for_var_tree(unsigned int string_id, Environment e,
    Node t) {
  Node data_actrecord_tree;

  // If t is empty, initialise it as gs->data_stack
  if (t == NULL) {
    t = get_top_data_actrecord_from_stack_tree();
  }

  if (variable_in_environment(string_id, e)) {
    // Add the 'data' field: data_stack->data
    // Cast the current tree (gs->prev->prev->...->data_stack)
    // Then add field access [cast-exp].fieldname
    return get_fieldaccess_tree(get_cast_for_data_actrecord_tree(e, t),
        newLeafId(get_string(st, string_id)));
  } else if (e->parent != NULL && (e->structname || e->type == FNBODY)) {
    // Must add in a previous link here...
    return get_data_actrecord_scope_for_var_tree(string_id, e->parent,
        get_prev_data_actrecord_scope_tree(t));
  } else if (e->parent != NULL && e->structname == NULL) {
    // If there is no structure allocated for this environment, it cant be on the
        data stack, so dont need to add a prev link
    return get_data_actrecord_scope_for_var_tree(string_id, e->parent, t);
  } else {
    perror("string_id doesn't exist in any environment");
  }
}

/**
 * DATA STACKLESS TRANSFORMATION
 */

/**
 * Transform usage of a variable at specific node in the specified environment.
 */
Node var_usage(Node t, Environment e) {
  string identifier, var_def_structname;
  unsigned int identifier_id;
  Environment var_def_env;
  Node return_node;

  assert(t->type == LEAF && t->subtype == IDENTIFIER);

  // Need to find the scope of the identifier
  identifier = getTokenLexeme(getLeafToken(t));
  identifier_id = get_string_id(st, identifier);

  var_def_env = get_environment_for_variable_definition(identifier_id, e);

  if (var_def_env == NULL) {
    // This is not actually a variable, just an identifier. Don't replace anything.
    printf("Environment of variable definition is null\n");
    return_node = t;
  } else {
    return_node = get_data_actrecord_scope_for_var_tree(identifier_id, e, NULL);
    set_transform_lock_node(return_node);
    set_var_usage_node(return_node);
    add_prev_data_levels_to_return_val(return_node, indirection);
  }
```

```
  return return_node;
}

/**
 * Build and return tree with any global variables initialised in appropriate
     global structure.
 */
Node get_global_variable_initialisations_tree(Node t, Node global_var_list,
    Environment e) {
  Node left_init = NULL, right_init = NULL, new_node = NULL;

  if (t->env) {
    printf("Changed environment\n");
    e = t->env;
  }

  if (t->type == DECL && e->parent == NULL) {
    if (t->right && t->right->type == INIT) {
      // Have a global var
      if (t->right->right) {
        // Global var initialises a value. This should be moved to the top of the
            main function
        printf("\t\tThis decl var is at global level\n");

        new_node = decl_usage(t, e);
        if (new_node != NULL) {
          // Remove this global variable - it is now represented in the global
              struct
          set_deleted_node(t);
        }
      }

      // If this is still a global variable, but not initialised, still remove it
      if (!is_struct_union_typedef(t->left)) {
        set_deleted_node(t);
      }
    }
    return new_node;
  } else {
    // Recurse right first to make it easier to match actual param name below.
    if (t->right) {
      right_init = get_global_variable_initialisations_tree(t->right,
          global_var_list, e); // Recurse right
    }
    if (t->left) {
      left_init  = get_global_variable_initialisations_tree(t->left,
          global_var_list, e);  // Recurse left
    }
    return merge_stm_lists(left_init, right_init);
  }
}

/**
 * Perform transformations
 */
void a_transform(Node t, Environment e) {
  bool new_env = false, replace_flag = false;
  Node new_node, insert_node;

  static Node root = NULL; // store root node of tree

  // Store location of root
```

```
if (root == NULL) {
  root = t;
}

// Track current environment
if (t->env) {
  printf("Changed environment\n");
  new_env = true;
  e = t->env;
}

// Process node depending on the type
if (t->type == STRUCT || t->type == UNION || t->type == STRUCT_DECL) {
  // Structures etc do not require processing.
  printf("Aborting on %s\n", get_node_type_string(t->type));
  replace_flag = true;
} else if(has_transform_lock_node(t)) {
  // Do not process any nodes which have a transform lock on them
  printf("Aborting on transform lock\n");
  replace_flag = true;
} else if (t->type == ';' && t->left->type == FN_CALL &&
    !has_transform_lock_node(t->left)) {
  printf("got solitary fncall");
  handle_fncall(root, t, e);
  replace_flag = true;
} else if (t->type == ';' && !has_transform_lock_node(t->left) &&
    is_nested_fncall(t->left)) {
  printf("got nested fncall");
   handle_nested_fncall(root, t, e);
  replace_flag = true;
} else if (t->type == DECL && e->parent == NULL) {
  // Look for a global variable declarations + initialisation
  if (t->right && t->right->type == INIT) {
    // Ignore for now
  }
} else if (t->type == LEAF && t->subtype == IDENTIFIER) {
  // Variable usage. Replace with heap location
  printf("Got variable usage: %s\n", getTokenLexeme(getLeafToken(t)));
  new_node = var_usage(t, e);

  if (new_node != NULL) {
    if (e->parent == NULL) {
      printf("\t\terror: This var is at global level\n");
    } else {
      swap_nodes(t, new_node);
      replace_flag = true;
    }
  }
} else if(t->type == CMPD) {
  // When entering a block, declarations appear in a list on the left of the tree
  // Statements appear in a list on the right of the tree
  // Process the addition of the struct first, then handle declarations...

  // Process declarations
  if (t->left) {
    if(t->left->type == DECL || t->left->type == DECL_LIST) {
      printf("Got DECL usage\n");
      new_node = decl_usage(t->left, e);
      if (new_node->type != STM_LIST && new_node->type != ';') {
        new_node = newNode(';', new_node, NULL);
      }
```

```
        // The initialising of variable values should be done first, so add them to
            the end of the STM_LIST
        insert_node = locate_end_of_list(STM_LIST, t->right);
        if (insert_node == NULL) {
          printf("STM_LIST null\n");
          new_node = newNode(STM_LIST,
                              new_node,
                              newNode(';', shallow_node_copy(t->right), NULL));
              swap_nodes(t->right, new_node);
        } else {
              printf("STM_LIST not null\n");
              insert_node->left = newNode(STM_LIST,
                                          new_node,
                                          insert_node->left);
        }
      }
      // Remove the declarations
      t->left = NULL;
    }

    // Entered a compound block: create the struct required (if present for this
        environment)
    if (t->env && (t->env->type == FNDEF || t->env->type == FNBODY)) {
      printf("Got environment create\n");
      new_node = get_push_data_actrecord_with_struct_tree(t->env);

      // Insert node to create and push new data frame onto data substack at
          beginning of block
      insert_node = locate_end_of_list(STM_LIST, t->right);
      if (insert_node == NULL) {
        printf("STM_LIST null\n");
        new_node = newNode(STM_LIST,
                            new_node,
                            newNode(';', shallow_node_copy(t->right), NULL));
        swap_nodes(t->right, new_node);
      } else {
        printf("STM_LIST not null\n");
        insert_node->left = newNode(STM_LIST,
                                    new_node,
                                    insert_node->left);
      }
    }
  } else if (t->type == DECL || t->type == DECL_LIST) {
    //printf("!!! %s\n", getTokenLexeme(getLeafToken(t->right->left)));
  } else if (t->type == RETURN) {
    // Only process return statements with a value
    if (t->left) {
      // Return value is in t->left
      a_transform(t->left, e);

      // Only do this for non-main fns
      if (get_current_fne(e)->string_id != get_string_id(st,
          MainFunctionIdentifier)) {
        insert_node = get_data_actrecord_scope_for_return_tree(e, NULL);

        // Insert data act record into tree
        new_node = newNode(';',
                            newNode('=', insert_node, shallow_node_copy(t->left)),
                            NULL);

        swap_nodes(t, new_node);
        // Insert pop data act record into tree
```

```
        insert_node = insert_node_after_node(get_pop_data_actrecord(), t, root);
        set_return_value_node(insert_node);

        printf("handled return statement\n");
        replace_flag = true;
      }
    } else {
      insert_node_before_node(get_pop_data_actrecord(), t, root);
    }
  } else if (t->type == FNDEF) {
    // Make all fn return types void. Rrturn type is on left of fnheader (which is
        t->left)

    // Fn identifier is on left of F (which is t->right of fnheader)
    if (get_string_id(st, getTokenLexeme(getLeafToken(t->left->right->left))) ==
        get_string_id(st, MainFunctionIdentifier)) {
      // Do not change the main function
    } else {
      // Return type is set to void
      t->left->left = newNode(DECL_SPEC_TYPE, newLeaf(VOID, NULL), NULL);
      printf("changed fn return type to void\n");

      // Remove all parameters. They are on right of F
      t->left->right->right = NULL;
    }
  }

  // If something has been replaced, do not recurse on tree.
  if (!replace_flag) {
    if (t->left) a_transform(t->left, e);
    if (t->right) a_transform(t->right, e);
  }

  // Track current environment - revert to parent.
  if (new_env) {
    e = e->parent;
  }
}


/**
 * Construct and return tree to load values into a structure for a function call
     (matching formal and actual params)
 */
Node get_load_values_for_struct_for_fn_tree(Node root, FnEntry fne, int *var_index,
    Node arg_tree, Environment e) {
  // Need to match each value being passed in (actual param) with the corresponding
      formal params

  Node left_params, right_params, loaded_param;
  Variable formal_param_var;
  string formal_param_name;

  //static unsigned int var_index = 0;

  if (arg_tree->type == ARG_LIST) {
    // Recurse right first to make it easier to match actual param name below.
    right_params = get_load_values_for_struct_for_fn_tree(root, fne, var_index,
        arg_tree->right, e); // Recurse right
    left_params  = get_load_values_for_struct_for_fn_tree(root, fne, var_index,
        arg_tree->left, e);  // Recurse left
```

```
    return merge_stm_lists(left_params, right_params);
  } else {
    // Have an arg. Get corresponding variable, then actual param name
    formal_param_var = get_variable_by_offset_from_varlist(fne->varlist,
        (*var_index)++);
    // formal_param_var = get_variable_first_from_varlist(varlist);
    formal_param_name = formal_param_var->string;

    // Value is this tree
    a_transform(arg_tree, e);

    loaded_param = get_load_param_into_struct_tree(fne->fndef->env,
        formal_param_name, arg_tree);

    //handle_nested_fncall(root, loaded_param, e);
    return loaded_param;
  }
}

/*
 * Initialise global stack + global variable initialisations
 */
void add_global_struct_to_tree(Node root) {
  Node main_fn_body, global_stack_var, global_stack_init, global_struct_push,
      global_struct_pop_node;
  Variable globalstack_var;

  // Locate the main fnbody
  main_fn_body = get_fnbody_for_fn_tree(root->env, MainFunctionIdentifier);

  // Create globalstack global variable
  global_stack_var = get_new_type_var_tree(GlobalStackType, GlobalStackIdentifier);
  // add_var_pointer(global_stack_var);
  add_storage_spec(global_stack_var, STATIC);

  // Insert global variable into tree
  insert_at_start_of_type(EXTERNAL_DEF, root, global_stack_var);

  // Insert global variable into varlist
  globalstack_var = newVariable(get_type_list_for_decl(st, global_stack_var->left),
                                lookup_string(st, GlobalStackIdentifier),
                                GlobalStackIdentifier,
                                global_stack_var,
                                global_stack_var->right->type);
  root->env->varlist = newVariableList(globalstack_var, root->env->varlist);

  // ---------------
  // PUT ITEMS INTO BEGINNING OF MAIN IN REVERSE ORDER

  // Push global struct onto stack at beginning of main
  global_struct_push = get_push_data_actrecord_with_struct_tree(root->env);
  insert_at_start_of_type(STM_LIST, main_fn_body->right->right, global_struct_push);

  // Create initialisation for main fn
  global_stack_init = get_globalstack_initialisation_tree();

  // Insert var init in main body - use main_fn_body->right->right since CMPD on
      RHS, then decl's lie on left and stm's on right (and decls get removed!)
  insert_at_start_of_type(STM_LIST, main_fn_body->right->right, global_stack_init);

  // ---------------
```

```
  // Just before returning from main, we should pop the global struct from the data
      stack
  global_struct_pop_node = get_pop_data_actrecord();
  insert_node_before_node(global_struct_pop_node,
      main_fn_body->right->right->right, root);

}

/**
 * CALL STACKLESS TRANSFORMATION
 */

/**
 * Convert all functions to be stackless (call stackless)
 */
void process_all_fncalls(Node t, Environment e) {
  bool new_env = false;
  static Node root = NULL;

  // Store location of root
  if (root == NULL) {
    root = t;
  }

  // Track current environment
  if (t->env) {
    printf("Changed environment\n");
    new_env = true;
    e = t->env;
  }

  if (t->type == ';' && t->left->type == FN_CALL &&
      !has_transform_lock_node(t->left)) {
    printf("got solitary fncall");
    handle_fncall(root, t, e);
  } else if (t->type == ';' && !has_transform_lock_node(t->left) &&
      is_nested_fncall(t->left)) {
    printf("got nested fncall");
    handle_nested_fncall(root, t, e);
  } else {
    if (t->left)  process_all_fncalls(t->left, e);
    if (t->right) process_all_fncalls(t->right, e);
  }

  // Track current environment (revert to parent)
  if (new_env) {
    e = e->parent;
  }
}

/**
 * Handle a (solitary) fn call
 */
void handle_fncall(Node root, Node t, Environment calling_env) {
  unsigned int string_id, var_index = 0;
  string fn_name;
  FnEntry fn_entry;
  Environment fn_param_env;
  Node fn_call, param_struct_push_node, param_struct_pop_node,
      params_load_to_struct_node;

  if (t->type == ';' && t->left->type == FN_CALL) {
```

```
    fn_call = t->left;

    // Get fn name, and extract corresponding FnEntry & environment
    fn_name = getTokenLexeme(getLeafToken(fn_call->left));
    string_id = get_string_id(st, fn_name);
    if (string_id == 0) {
      // fn name wasn't found. Fn wasn't defined in this source file. Do not
          transform.
      return;
    }
    fn_entry = get_fnentry_by_stringid(root->env->fnlist, string_id);
    fn_param_env = fn_entry->fndef->env;

    // Push struct onto stack
    param_struct_push_node = get_push_data_actrecord_with_struct_tree(fn_param_env);

    insert_node_before_node(param_struct_push_node, t, root);

    // Process parameters
    if (fn_call->right) { // fn call has parameters. process each.
      // Get tree of params loaded into struct & insert before fn call
      params_load_to_struct_node = get_load_values_for_struct_for_fn_tree(root,
          fn_entry, &var_index, fn_call->right, calling_env);
      assert(params_load_to_struct_node != NULL);
      insert_node_before_node(params_load_to_struct_node, t, root);

      // Remove fn call params
      fn_call->right = NULL;
    }

    // Handle return value (env is attached to fndef->fnbody->cmpd)
    //get_return_value_from_struct_tree(fn_entry->fnbody->right->env);
    //get_return_value_from_struct_tree(fn_entry->fndef->env))

    // Pop act record
    param_struct_pop_node = get_pop_data_actrecord();
    // Insert_node_after_node(param_struct_pop_node, t, root);
    insert_node_after_node(param_struct_pop_node, t, root);

    printf("fn call processed - %s\n", fn_entry->fnname);
  }
}

/**
 * Convert a (solitary) fn call to stackless
 */
void stackless_fn_call(Node t, Environment e) {
  static unsigned int unique_label_id = 2000;
  static Node root = NULL;
  bool new_env = false;
  FnEntry current_fne, calling_fne;
  string return_label_name, fn_name;
  char tmp_buf[BUFFER_LEN];
  unsigned string_id;
  Node return_label, store_return_address, asm_call, next_stm, curr_stm, asm_return;

  memset(tmp_buf, '\0', BUFFER_LEN); // Initialise buffer

  // Store location of root
  if (root == NULL) {
    root = t;
  }
```

```
// Track current environment
if (t->env) {
  printf("Changed environment\n");
  new_env = true;
  e = t->env;
}

if (t->type == FN_CALL) {
  printf("stackless fncall start\n");

  current_fne = get_current_fne(e);
  strncpy(tmp_buf, current_fne->fnname, BUFFER_LEN-4);
  strncat(tmp_buf, itoa(unique_label_id++), 4);
  return_label_name = String(tmp_buf);

  // Get fn name, and extract corresponding FnEntry & environment
  fn_name = getTokenLexeme(getLeafToken(t->left));
  string_id = get_string_id(st, fn_name);
  if (string_id == 0) {
    // fn name wasn't found. Fn wasn't defined in this source file. Do not
        transform.
    return;
  }
  calling_fne = get_fnentry_by_stringid(root->env->fnlist, string_id);

  curr_stm = get_current_stm_node(root, t);
  assert(curr_stm->type == ';');
  next_stm = get_next_stm_node(root, curr_stm);

  // Get trees for return label, storing return address and asm jump for this
      function
  return_label = get_label_tree(return_label_name, next_stm);
  store_return_address =
      get_push_call_actrecord_with_return_tree(return_label_name);
  asm_call = get_asm_call_tree(calling_fne);

  // Insert above trees into appropriate locations in tree
  insert_node_before_node(store_return_address, curr_stm, root);
  swap_nodes(curr_stm, asm_call);
  swap_nodes(next_stm, return_label);

  printf("stackless fncall end\n");
} else if (t->type == RETURN || is_return_value_node(t)) {
  // Insert return statements into tree to revert control to location stored on
      top of call stack
  if (get_current_fne(e)->string_id != get_string_id(st, MainFunctionIdentifier))
      {
    asm_return = get_asm_return_tree();

    if (t->type == RETURN) {
      swap_nodes(t, asm_return);
    } else {
      insert_node_after_node(asm_return, /*curr_stm*/ t, root);
    }
    printf("stackless return end\n");
  }
} else {
  // Recurse left and right
  if (t->left)  stackless_fn_call(t->left, e);
  if (t->right) stackless_fn_call(t->right, e);
}
```

```
  // Track current environment (revert environment)
  if (new_env) {
    e = e->parent;
  }
}
```

Listing A.5: Environment header (environment.h)

```
#ifndef ENVIRONMENT_H_
#define ENVIRONMENT_H_

/**
 * Environment.h
 * Represents the scopes that exist in a program.
 * Forms a singly linked list.
 */

#include <stdbool.h>
#include "variable.h"
#include "fnentry.h"
#include "string.h"
#include "typelist.h"

typedef struct environment_ {
  int                         type;           // Stores type of environment
      (CMPD, FNDEF, etc)
  struct fnentry_             *fne;           // Corresponding fn entry structure
      for a function definition scope.
  struct variable_list_       *varlist;       // List of variables defined in
      scope.
  struct typedef_list_        *typedeflist;   // List of typedefs in file.  Only
      used for initial environment.
  struct fnentry_             *fnlist;        // List of functions in file. Only
      used for initial environment.
  struct environment_         *parent;        // Link to previous environment.
      Null for initial env.
  string                      structname;     // Name of structure representing
      this scope.
  bool                        has_return_field;// Flag for return field in this
      scope.
} *Environment;

/**
 * Constructor for environment. Requires parent environment & type.
 */
Environment newEnvironment(Environment parent, int type);

/**
 * Add a variable to environment.
 */
void add_variable_to_env(Environment e, Variable v);
/**
 * Add a variable list to environment. Used to add a list of parameters, or bulk
     variable declaration.
 */
void add_envlist_to_env(Environment e, VariableList el);
/**
 * Add a typedeflist to environment.
 */
void add_typedef_to_env(Environment e, TypedefList tdl);
/**
```

```
 * Add a fnEntry to environment.
 */
void add_fnentry_to_env(Environment e, FnEntry fne);

/**
 * Set the fnEntry for this environment.
 */
void set_fnentry_for_env(Environment e, FnEntry fne);

/**
 * Dump the contents of this environment.
 */
void env_dump(StringTable st, Environment e);

/**
 * Determine if a variable with specified string id is in specified environment.
 */
bool variable_in_environment(unsigned int string_id, Environment e);

#endif /*ENVIRONMENT_H_*/
```