Total Pasta:
Static Analysis For Unfailing Pointer Programs

Neil Mitchell

Supervisor: Professor Colin Runciman

Fourth year project report submitted May 5, 2004 toward the degree of MEng Computer Science and Software Engineering from the Department of Computer Science, at the University of York.

Number of words = 28183, as counted by the Unix `wc` command on the LaTeX source. This includes the body of the report, but excludes the appendices.

Page count = 66, including the body of the report, but excluding the appendices.

Abstract

Most errors in computer programs are only found once they are run, which results in critical errors being missed due to inadequate testing. If additional static analysis is performed, then the possibility exists for detecting such errors, and correcting them. This helps to improve the quality of the resulting code, increasing reliability.

In this project the existing static analysis research is reviewed, along with implementations used both by normal programmers, and used in safety critical applications. A static analysis program is then designed and implemented for the experimental pointer based language Pasta.

The resulting program checks for totality, proving that a particular procedure cannot crash and will terminate. Where a procedure does not satisfy this, the preconditions for the procedure are generated.

Contents

Chapter 1:

Introduction

This chapter covers the motivation and aims behind the project, and gives a brief summary of the following chapters.

1.1 Motivation

Many computer programs crash at run time, and often extensive testing fails to catch subtle and obscure bugs. Static analysis [23] allows certain properties of a program to be calculated without running the program, which can highlight previously unseen errors in computer code. The analysis of programs which directly manipulate pointers is a particularly complex area.

The Pasta programming language [25] is an experimental pointer programming language that can be checked using graph theory to prove properties about the manipulation of data structures. For this graph analysis to be valid, every procedure in the program must be safe, and produce an output for every input. To prove this in the Pasta programming language it is to prove that the program does not crash, and that it does not loop infinitely.

1.2 Aims

The first aim of this project is to research the existing static analysis tools and methodologies available, and to determine their suitability in respect of the Pasta programming language. The second aim is to produce a suitable analysis engine for Pasta that can correctly assert that a function within a given program is total.

1.3 Chapter review

Chapter 2 covers the background theory on pointers and the Pasta programming language. Chapter 3 reviews existing static analysis theory and implementations. Chapter 4 covers the initial design decisions, and detailed discussion of the Pasta language. Chapter 5 describes how a forward analysis engine could be designed. Chapter 6 covers the design of a backward analysis engine. Chapter 7 describes a concrete implementation of the system designed in chapter 6, including information such as programming language, and how certain concepts can be implemented. Chapter 8 gives results obtained by the implemented program, including areas where it fails to perform optimally. Chapter 9 is the conclusion and covers an appraisal of the current system, along with areas for possible improvement.

Chapter 2:

Background

This chapter covers the background material needed to understand the remainder of the project. The subject of pointers, and in particular their place within the Pasta language, is covered. A knowledge of discrete mathematics to first year degree level is assumed, along with familiarity with some programming languages, such as C[5].

2.1 Pointers

Many computer programs use an area of memory known as the heap to store pieces of data. Often large arrays and dynamically allocated data structures are placed on this heap. In order to refer to a piece of memory on the heap programs use pointers, which are variables that point to a location within the heap, hence the name.

Many computer programming languages limit the types of operation that can be performed on pointers, giving different levels of power. Pointers allow many advanced computation features, however with this power comes an additional class of programming errors that can be made.

2.1.1 Addresses

Each location in the heap corresponds to an address in the system's memory, and this can address can be dereferenced to obtain the contents at this point. Internally addresses are stored in the same way as integers, therefore it is possible to perform numeric operations such as addition, subtraction, bit-wise inversion and others on an address. As a result of this, an address is not guaranteed to point to a location within the heap, or even within addressable memory. If the address does point to within the heap, then it does not necessarily point at an object, but may point into either an unutilised area of the heap, the middle of any given object within the heap, or an object that has since been deallocated.

The advantage of allowing the power of addresses is that it allows higher performance code to be written, and is essential for low level system programming. Some languages that expose the entire power of addresses include Assembly language and the C programming language. While the use of direct addresses within C programs is discouraged, there are some more common uses. Often to enumerate through an array the pointer to the first element is increased, by adding to the pointer the size of each element. Another use for addresses is when storing doubly linked lists, the pointer to the previous item and the next item can be coalesced into one pointer field with an XOR operation. When reaching an item, the program will know either its previous or next element, and by performing XOR will hence find the other one. This allows space to be saved.

2.1.2 Restricted Addresses

One method used to restrict the danger in using addresses is removing the ability to perform mathematical operations is removed. All values used as pointers are then determined by calls to memory allocation operations. After a pointer has been initialised, it will point at a valid object, unless that object is subsequently freed.

An example of a language that supports restricted addresses is Pascal [18], which disallows some dangerous operations, and still allows the programmer to manually manage the allocation and deallocation of memory.

2.1.3 References

A reference is a far stronger construct, and requires that every pointer variable points at a valid type of object. A pointer cannot become orphaned, or any other invalid value, so every pointer dereference is guaranteed to be safe. Most languages which have reference types also have some special reserved value to represent "no pointer". In Java [16] this is called `null`. All pointers are typically initialised to this reserved value, and when they are assigned to a pointer value, that pointer value is guaranteed to remain correct while that pointer is assigned to it. These languages usually give some form of automatic memory management, such as garbage collection in Java or reference counting in Visual Basic [11].

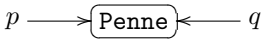
While less power is exposed than in other languages, there are far higher safety guarantees. Teaming references with checked array access (which is in effect what references are) gives the assurance that memory contents cannot be accidentally overwritten. This is essential when a language needs to assert some safety criteria – for example the Java programming language does not perform unchecked memory operations.

2.1.4 Aliasing

When two pointers refer to the same heap cell, they are said to be aliased to each other. This has the effect that if the underlying heap cell is mutated, then the effect of this will be visible from both pointers. This is easiest to illustrate with the following example written in C:

```
char* p = "Pasta";
char* q = p; //q and p are now aliased
strcpy(p, "Penne"); //change the value of p
```

After this code is executed, the observed value of `q` will have changed from `"Pasta"` to `"Penne"`. Diagrammatically an alias can be represented as:



2.2 Pasta

The Pasta programming language [25] is designed to represent pointer manipulating programs, to support practical research experiments. The language is small and simple, including only operations that are useful for research experiments, not writing large scale software. The syntax of Pasta is based on that of the C programming language, however the semantics differ greatly.

2.2.1 Initial Example

While all C style programming languages are introduced with the classic "Hello, World!" program, the equivalent in pointer languages would probably be insertion into an ordered singly-linked list without duplicates. To properly introduce the concepts behind Pasta, this is given as an example, in several portions.

The first section of any Pasta program is the signature, which comes at the top of each file. A signature is given below:

```
list {
    nil();
    cons(int head, ptr tail);
}
```

This statement defined a type called `list`, with two subtypes being `nil` and `cons`. The `cons` subtype has fields called `head` which is an integer, and `tail` which is a pointer. Any `ptr` field can point to any object on

the heap. From this the representation of a linked list can be constructed, with a list comprising of successive `cons` structures, the next element being pointed at by a `tail`, and a final `nil` element. To obey the ordered property it is also the case that `x->head` is numerically less than `x->tail->head`, assuming the `head` and `tail` selectors are valid.

```
-- inserts an element into a list
-- requires the initial list be sorted
-- ensures the final list is sorted
insert(int i, ptr s) {
    while (s::cons && s->head < i) s = s->tail;
    if (s::nil || s->head > i) *s = *cons(i,copy(s));
}
```

This function traverses down the list using the `while` statement, at each stage checking that the end of the list has not been reached. The `s::cons` expression checks the subtype of the variable `s`, and is true if its subtype is `cons`. At each stage the pointer `s` is advanced down the list, with `s = s->tail`. At the end, the heap cell pointed at by `s` is overwritten with an object of subtype `cons`, with the `*` operator performing in-place assignment. This means that the previous node's `tail` pointer does not need to be updated, as the new node is at the position of the old node.

The final piece of code to complete this sample program would be a distinguished `main` procedure, that is executed upon startup.

```
main() {
    ptr r = nil();
    insert(1,r); insert(9,r);
    insert(2,r); insert(8,r);
}
```

This procedure inserts the elements 1, 9, 2 and 8 into the linked list pointed at by `r`. After executing this procedure, the heap could be visualised as:

$r \longrightarrow 1 \longrightarrow 2 \longrightarrow 8 \longrightarrow 9 \longrightarrow nil$

2.2.2 Typing

In Pasta there are two fundamental types of item, integers (using the `int` keyword), and pointers (using `ptr`). An item that is declared as an integer stores a number, and this number may be compared with other numbers. Pointers are reference types, and in contrast to Java, there is no `null` keyword, so all pointers refer to valid heap objects as described in the type signature.

The Pasta interpreter performs a limited static analysis on the program before execution, checking that all assignment occurs between like types, and that comparison operators other than equality and inequality are done only on numbers. In addition, there are some operations than can only be performed on pointers (`::` and `->`), and these are validated as well. No checks are made statically of the subtypes, so the `->` operation may fail at run time.

More than one subtype may have an identically named selector, and the appropriate field is selected at runtime, based on the subtype. All selectors of the same name must have the same type, allowing the type

analysis to be performed with relative ease. This overloading can be thought of as similar to the C++ [6] `virtual` keyword.

2.2.3 Assignment

Pasta permits destructive assignment, where a pointer is modified to point at another location on the heap. Pasta also supports starred assignment, whereby the contents of a memory cell pointed to by a pointer are overwritten directly. When a pointer value is first declared, it must be assigned an initial value, for example `r` in the procedure `main`.

The numeric assignment in Pasta is quite different to this, and all `int` variables are unrelated. When an assignment is made, changes to one variable do not affect others.

2.2.4 Goals of Pasta

The Pasta language can be transformed into a graph representation, and this representation can be used to check that the program does what is intended. The graph analysis treats the program as a mathematical function, taking an input data structure to an output structure. A prerequisite for the graph analysis is that the procedure successfully reaches generates an output in all configurations. This condition needs to be checked by another form of analysis, and this project attempts to achieve this.

The Pasta language can also be compiled into functions useable from a C program. Routines manipulating pointer data structures can be written in Pasta, with assurances about safety not normally associated with C.

2.3 Total function

Since the requirement of static analysis on Pasta is that all the functions are total, it is useful to first define what a total function is. More details can be found in an introductory computation textbook, such as [22].

A total function is a relation that uniquely associates members of an input set with an output set, and is valid for every member of the input set.

$$\forall x \in \mathrm{In} \bullet f(x) \in \mathrm{Out}$$

When talking about computer programming languages totality maps directly onto the idea that the procedure must complete, for example by not crashing. While mathematics cannot enter infinite computation, this is not the case for a programming language. If a routine in a programming language never terminates then this is equivalent to not producing any value, and hence breaks the totality of the function.

2.3.1 Deterministic Output

For a true function, the output must be based solely on the input. This is not the case with most procedural programming languages, as often these programs interface with the rest of the computer. Examples of this include user interaction, for example the C `scanf` statement, or file accesses. Other examples of how the output could not be based on the input include access-time or date-based routines, global variable use, or use of a random-number generation. The Pasta language has no functions defined which are not deterministic, hence this problem is avoided.

At a lower level, Pasta does have operations that depend on the state of more than just the function input. When a new value is created on the heap, the location at which it is placed is based on what is already on the heap. Fortunately, details such as these are not propagated up to the programmer, and the program cannot behave differently based on the layout of the heap.

2.3.2 Termination

The Halting Problem [27] states that for any Turing Complete language, the problem of termination is undecidable in the general case. This means that it is not possible to write a computer program that takes as an input a definition of a programmed function, and returns as an output whether it terminates or not.

It is however, possible for certain types of function to check for termination. For example, if a function has no loops, and no further function calls then it is easy to determine that it terminates. If the program consists only of a loop that is guarded by a tautology, then it is can be determined that it loops infinitely. A number of these special case termination arguments exist, and can be used to compute termination for a subset of possible programs.

2.4 Summary

This chapter has shown an example of the Pasta programming language, along with basic specifications of its use. The purpose of the analyser has been set out, including what does not need to be checked. The concept of a total function is defined, along with its equivalent for a computer program.

Chapter 3:

Static Analysis

This chapter covers the topic of static analysis, its overall aims, and various methods of implementation. Existing analysis programs are covered, along with how they can be used and what guarantees they give. Various methods of representing pointer data structures are also covered.

Static analysis a term used to describe the analysis of a program, and determining properties about it, without executing the code directly. It analyses all branches of the code, and the analysis itself is guaranteed to terminate. Static analysis is used for many purposes, including checking for safety of programs, and for generating higher performance software.

Often in static analysis it is not possible to completely determine a property. In this circumstance, the analysis can either ignore a potential problem, or generate a possibly false warning. The choice of which to do depends on the purpose of the static analyser.

There are many different models of static analysis, some of the more common ones are now described. More information on this topic can be obtained from sources such as [23].

In order to show the difference between the various following approaches, I will use the example of calculating $z = x \times y$ without the use of multiplication, where all the variables are positive integers, and x is destroyed. This can be done by the following code:

```
[z = 0]^1;
while ([x > 0]^2) {
    [x = x - 1]^3;
    [z = z + y]^4;
}
```

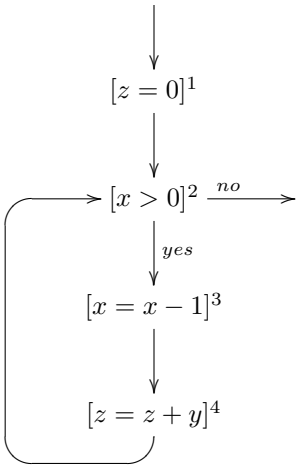
3.1 Data Flow Analysis

For data-flow analysis, the program can be considered as a graph. Each statement in the program then corresponds to a node, and the flow between the statements as directed arcs on this graph. Program statements such as `if` and `while` lead to situations where two arcs either arrive at or leave a single node. The example given above would translate into a graph as shown in Figure 3.1.

Using the graph representation, equations are then set up to define the reaching definitions, where a variable obtains its value from. Separate definitions are provided for the entry and exit to each node, allowing the output to be modelled in terms of the input. For example, statement 2 does not modify the state so $RD_{entry}(2) = RD_{exit}(2)$. Before the program is executed, on entry to statement 1, none of the values have a known reaching definition so $RD_{entry}(1) = \{(x, ?), (y, ?), (z, ?)\}$. However, after statement 1 has been executed $RD_{exit}(1) = \{(x, ?), (y, ?), (z, 1)\}$. Note that the value associated with z is 1, corresponding to the statement that set it, and not the value it was set to.

After these reaching definitions have been created, they are then solved to determine where a variables value at any point comes from. This information can then help show what the value of the variable is, and how it changes. In the course of solving the equations, it may be necessary to decrease the precision of the analysis, to ensure termination.

Figure 3.1: Multiplication example as a graph



3.2 Constraint Based Analysis

Constraint based analysis uses a similar graph representation to data flow analysis, however it focuses on the node instead of its entry and exit points. Each statement is reformulated as a constraint, describing the effect of the node. These constraints are then solved using general constraint solving techniques. This type of analysis seems to be more common when working with functional programs, and vast simplifications rely on the the absence of side effects.

The example given for data-flow analysis could be reformulated as a constraint based problem. The exit from the first statement can be represented as $RD_{exit}(1) \supseteq \{(z, 1)\}$. Note the use of $\supseteq$ as a constraint, many of these can be generated and then solved to find a fixed point.

3.3 Forward Analysis

In this method the program is executed on a generic state, which represents all the possible states the program could be in just after a particular statement. As the statements are executed, the state is modified, and the end state can be used to deduce properties on. Where a statement can be executed more than once, for example statement 3, the generic state must cater for all these possibilities.

Finding suitable abstractions for each variable is often specific to particular problems. One method often used for numeric variables is range analysis, determining which variables may be between which values. For example, after statement 1 has been executed $0 \leq z \leq 0$, while all other variables are unknown.

3.4 Backwards Analysis

Backwards analysis takes a postcondition at the end of the procedure, and transforms it over all the statements in reverse order to obtain a precondition. This is particularly useful when the exact properties required at the end can be determined before analysis starts.

The disadvantage of using backward analysis compared to the other methods is that sometimes it becomes counter intuitive. When an error is reached, it is not always easy to pinpoint where the error is – only that

it exists. On the other hand, forward analysis is generally able to give the precise statement which caused the error.

Using backward analysis on the above example, if the postcondition for statement 4 was that $z \geq 0$ then the precondition would be $z + y \geq 0$. When transformed over the entire program, $z \geq 0$ would become $y \geq 0$. However, taking the postcondition $z > 0$ would generate $y > 0 \wedge x > 0$ as a precondition to the function.

3.5 Programming Tools

The following programming tools based on static analysis are in widespread use by many programmers. In most cases the static analysis is performed with standard source code, with few modifications if any.

3.5.1 Type Checking

The most common type of static analysis is the type checking perform on statically typed languages at compile time. These include languages such as C++ [6] and Haskell [20]. In these languages there are lots of distinct types of objects, and different functions operate on different types, as pre-declared. The C++ language requires complete declarations of all types, and then uses these types to implement overloaded functions, where two functions of the same name but different type signatures can be created and used, with the compiler deciding at compile time the appropriate version. The Haskell language takes a different approach, with types being optional, and inferred via unification where they are absent. In both cases, at compile time, the syntax tree can be completely decorated with all the types of every variable present in the program.

3.5.2 Liveness Analysis

Another type of static analysis that is very common is liveness analysis [15]. The point of liveness analysis is to determine whether a variable is live at a particular statement in the program. A variable is live at a statement if its value is used before it is assigned to, meaning some part of the program requires the value as it is stored. One reason for liveness analysis is to help code optimisers. When compiling source code to a target processor, particularly a register-based processor, some variables are stored in memory and some are stored in fast access registers. With liveness analysis the variables stored in the limited number of registers can be chosen in a more efficient manner.

More modern compilers now tend use liveness analysis also for tracking uninitialised variables. When a variable is declared in a language such as C, its initial value is undefined, and use of this value can lead to unpredictable behaviour. Liveness analysis allows a warning to be given at compile time, and the error can then be corrected.

3.5.3 Compilers

The most common tool to analyse source code is a compiler. This section concentrates on some of the static analysis tasks performed by compilers, in addition to those mentioned above. For most examples, the GCC [24] compiler is used, as this is open source and is documented to a greater degree than commercial alternatives.

One example of static analysis is the `nonnull` attribute, which can be applied to a function. This means that passing a null pointer to the function is not a valid operation. For example, a definition of the common `memcpy` function might be augmented in the following way:

```
extern void*
memcpy(void *destination, const void *source, size_t length)
```

```
__attribute__((nonnull (1,2)));
```

This `nonnull` attribute tells the compiler to track the values passed to `memcpy`, and complain if either of them is a null pointer. This is done using static analysis, and a variation on the liveness analysis described above, that also tracks the values. This attribute is also used to give hints to the compiler, by assuming that the values passed in are not null.

Another use of static analysis in the code comes not from correctness checking, but purely from performance enhancing code. Many modern processor architectures, for example the Intel Itanium, allow hints to be given as to whether a branch will be taken or not. For example, in an `if` statement typically one branch will be taken more than the other. The compiler can emit this information to the assembly code allowing the processor to make the common case more efficient while still keeping the less common case correct. GCC can be configured to make these decisions based on information garnered from static analysis, principally variable-flow analysis. In addition to traditional static analysis, the compiler also has heuristics for common behaviour, such as more integers are positive than negative. This information is then propagated using static analysis to determine likely code paths. This is an example where the static analysis does not have to be correct, and indeed, provided it is correct slightly more than half the time, will result in a performance increase.

Most of the static analysis in compilers comes from examining variable flows, by tracking what gets assigned to what. These types of analysis are commonly broken by introducing values that have not been tracked up to that point, for example results for external functions or global variables. In this case, the static analyser usually keeps silent, and does not issue a warning of any sort, as these cases are very common.

3.5.4 Splint

The Splint tool [14], and various papers about it, relate to the design of a light-weight static analysis checker. Initially the tool was based on the concept of an improved Lint [19] (Lint being an early static analyser for C), but now Splint is more concerned with static analysis from a security point of view.

The goal of Splint is to highlight as many possible programming errors as possible. It is neither sound nor complete, producing both false positives and false negatives. The types of bugs that are focused on are buffer overflows and resource leaks, as these tend to be a security concern. Program correctness is not explicitly tackled, although many programming errors would give rise to errors that are flagged. Splint has various analysis tags that can be applied, such as `nonnull`, although in general the use of an annotated library, in contrast to an annotated program, is required to diagnose security problems. Splint operates at a procedural level, validating each procedure in isolation.

One interesting point about Splint is the treatment of loops. Splint has a list of common patterns, and then uses these to recognise structures in loops. For example, in C, the most common form of loop statement is probably `for (int i = 0; i < n; i++)`, where `i` typically indexes into a buffer. Splint uses this to determine that the buffer is accessed in positions 0 to `n`-1.

3.5.5 Stanford Checker

The Stanford Checker [13] is a static analysis tool specifically targeting the Linux kernel. In addition to errors like buffer overflows, the checker also targets instances where the internal kernel API has been violated – for example locks not being freed. This has now been incorporated into a commercial product called SWAT [12].

The analysis performed is both unsafe and incomplete, but still useable by practical programs. This tool analyses the complete C language, performs interprocedural analysis, and detects a large range of errors. The

tool also requires no additional annotations to the code, and instead relies on a technique termed statistical inference. In a codebase, where a statement `lock` is usually followed by an `unlock`, the tool infers that this is the correct behaviour. Then in future where this is violated, a warning is given.

The major project on which this analysis tool has been tested is the Linux kernel. The analysis tool was run, and found over 2000 bugs in the code. A rate of 1 false positive to every 4 to 5 real bugs is quoted, in contrast to around 1 real bug to 50-100 false positives for a tool such as Lint. Unfortunately the inner details are not available, but this analysis tool seems very powerful.

3.6 Formal Verification Tools

While the tools given above are used in many projects, and can be integrated into an existing project, none provides a guarantee of correctness. In fact the level of assurance offered by these tools is minimal, and for a higher level of static analysis a more formal approach is required. This will often require many additional annotations to the code, and either the use of a safe subset of a programming language, or using a more mathematical notation. These tools are typically only used when a high level of reliability is required, due to their high cost to integrate into the development process.

3.6.1 The B Method

The B Method [2], as implemented in the B Toolkit, is very similar in spirit and principle to the Z Notation [1]. Due to this similarity, only B is covered.

The basic principle of the B Method is that the state of the program is represented as a finite number of variables, which are defined in terms of mathematical objects such as numbers, sets and functions. A procedure then takes this state, and mutates it in some way, using both mathematical operators and destructive assignment. All the operations in B are defined in terms of predicate logic, all procedures have a pre-condition, and the entire state space has an invariant. The B Toolkit is then able to take this specification of the program, and for each procedure, prove that if the preconditions and the invariant hold before, then afterwards the invariant will hold. In B this is equivalent of checking that:

$$Invariant \wedge Precondition \Rightarrow [Statement]Invariant$$

In B, various constructs that are common in most programming languages are missing. Examples of these include procedure calls (both recursive and non-recursive) and any form of looping behaviour. This ensures that the program does not loop infinitely. The way the B toolkit proves properties about programs is by taking the postcondition, and transforming it in light of the statements – essentially backward analysis.

The B Method also provides a mechanism for turning a formal specification into a concrete implementation, without loosing the formality. This process is called reification, and involves reformulating from abstract maths to a lower level representation, in stages. At each stage it is possible to prove that the semantics of the program have not been changed, and that all the conditions still hold.

One step in the reification process involves introducing loops using the `WHILE` statement. The B Method requires that a `WHILE` contains a loop invariant and a loop variant, in addition to the statement to be executed and a conditional guard. The loop invariant must be true at the end of every loop execution, and can simply be specified as a tautology. The loop variant however is a variable, which must correspond to a positive integer value, and which must decrease every loop iteration. This ensures that only a finite number of iterations are possible, bounded by the value of the loop variant. Determining a loop variant must be done by the programmer, to do this automatically would be equivalent to solving the halting problem.

The B Method is a very rigourous tool for engineering software, using formal mathematics to prove correctness. Unfortunately, the use of such a toolkit is a complex undertaking, and standard programming code is

hard to construct in such a manner.

3.6.2 SPARK

In many ways the SPARK programming language [7] is in the same family tree as the B Method, yet at a much lower level of restriction and formality. SPARK uses a subset of the Ada programming language [4], plus appropriate annotations, to perform verification. When compared to the B Method it is interesting to observe what assurances are not provided in SPARK. Another useful exercise is to view what features of Ada have been removed, in order to simplify the verification.

As an introduction it is useful to show an Ada program, with SPARK annotations.

```
procedure Add(X: in Integer)
--# global in out Total;
--# derives Total from Total, X;
--# pre X + Total <= Integer'Last;
--# post Total = Total~ + X;
is
begin
   Total := Total + X;
end Add;
```

In a similar way to B, SPARK provides pre and post conditions that can be proved mathematically. SPARK also has annotations for data flow, which allows programming contracts to be enforced between areas of the program.

The primary goal of SPARK is to prevent software exceptions from being raised. In Ada, there are four classes of primitive exceptions, namely tasking errors, program errors, storage errors and constraint errors. The tasking and program errors correspond to features using the multi-threading mechanisms inherent in Ada, and these are disallowed by SPARK. The other errors have to be dealt with in other ways.

Storage errors are caused by the program running out of memory, and the is prevented by disallowing any operations that would require dynamic (i.e. heap-based) memory to be allocated. This means that all arrays must be of fixed size, no objects can be created on the heap, and also the use of recursion is forbidden. This ensures that the memory requirements of a program can be calculated statically, including the depth of the call stack, and with appropriate hardware resources can be always satisfied.

Constraint errors are caused when numeric values are out of their appropriate range, for example arithmetic overflow and invalid array-bounds access. These errors cannot be removed, and instead are proved. The proof is primarily performed using data flow analysis, which also allows the `derives` and `in`/`out` statements to be checked.

The handling of loops is of particular note in SPARK, all `loop` statements must be broken using an assertion. The analysis divides the loop into three separate conditions, each of which can be proved individually. When an assert is placed in the middle of a loop, the obligations are that the code from before the loop to the assert must be valid, from the assert completing one iteration of the loop and back to the assert, and from the assert statement to the end.

One additional limitation in SPARK is the removal of aliasing. In Ada points can also point at stack based data structures, however if the same piece of memory is referred to by more than one variable, then SPARK catches this and reports it as a major fault. The removal of aliasing allows drastic simplifications to be made to the analysis.

SPARK does not attempt to venture into the problem of termination. In Ada (unlike C), the `for` statement is always guaranteed to terminate, however the `loop` statement may enter an infinite cycle. This is detected by SPARK if the code matches some specific templates, however no proper analysis is used to determine if a loop terminates.

SPARK is a very powerful tool, and has been used in the automotive industry to produce software to a high degree of reliability [17]. At the end, the tool outputs a list of equations, that must be proved. The SPARK prover is able to discharge most of these obligations, but as with the B Method, significant effort must be put into proving the remaining obligations. SPARK requires less effort to use than the B Method, but in exchange provides weaker guarantees about the resulting program.

3.7 Static Analysis for Pointers

3.7.1 Analysis of Pointers and Structures

Analysis of Pointers and Structures [9] uses the source code from a program to automatically generate a description of the pointer structures it could generate. The features that make this paper stand out are that it generates a framework on which all possible data structures can be generated, and it places limits on the size of the generate structure.

The first stage of the process is the reduction of a program to its smallest atomic statements. For example, the statement `T2 = X.head.tail` is split into `T1 = X.head` and `T2 = X.tail`. Another example is the creation of a new `cons` node. While in LISP this would involve creating the node and assigning values to the constituent parts, this is split up into a create statement, and two assignment statements. As an example `T1 = cons(X, Y)` becomes `T1 = cons(); T1.head = X; T1.tail = Y`. This approach allows their analysis engine to work on a smaller range of inputs.

Pointer structures are represented by Shape Storage Graphs (SSG). This graph consists of nodes representing heap cells, and edges which point from variables to heap cells, and from heap cells to heap cells. More than one edge can leave each node, and this represents that the data structure may be in more than one state depending on the executed code. More than one SSG could represent any particular data structure, and the different graphs can encode different information and to different levels. The occurrence of cycles in the resulting graph can also be attributed to two factors, both a cyclic data structure and unbounded acyclic data structures.

Chase et. al. give two algorithms for generating such a representation from the source code. The main way of limiting the number of nodes created is by grouping nodes created at the same line of the program, and ensuring that all nodes created on different lines remain disjoint. This exploits the property that different parts of the program are likely to do different things. The more advanced method presented is more efficient, especially when the graph is sparse (most nodes are pointed at by only one reference), and studies have shown this to be a very common situation.

Finally the authors give details of how to expand procedures inline, although make no mention of the recursive situation. The problem with this method for their particular implementation is that some methods are helper methods, and are mainly syntactic sugar for common patterns, while others contain portions of logic. For recognising common code statements to generate disjoint nodes, it is likely that helper methods should be treated as separate lines of code, while logical methods should be grouped. How to achieve correct grouping of methods is left as an open problem.

3.7.2 A Safe Approximation Algorithm for Interprocedural Pointer Aliasing

A Safe Approximation Algorithm for Interprocedural Pointer Aliasing [21] discusses algorithms for determining which pointers may be aliased to which, including through procedures. The focus is on creating a May Alias relation which can determine whether it is possible that the two pointers are aliased. This relation is designed to report that the pointers may be aliases when it is unable to find proof to the contrary.

The authors tackle the problem for C programs, and note that this is much harder than the equivalent problem in FORTRAN, because of the side effecting operators and the C pointer model. In order to make the problem more manageable they exclude certain operations including type casts and function pointers. This restriction enables them to make certain assumptions that they require for their analysis. One important thing to note is that the analysis *does* handle arrays and pointer arithmetic, albeit in a naïve manner.

The central idea is that each cell on the heap is an object, and each item capable of pointing at an object has an object name. Object names are defined recursively as either a variable, the dereference of an object name, or an object name followed by a selector field. An alias can then be defined as when two object names point at the same object, and a list of unordered pairs can be maintained to keep this information.

This approach needs some way to limit the amount of space stored, otherwise a data structure like a linked list, with two variables referring to the head, would have an unbounded number of aliases. The way this is done is by choosing a limit, k, and then bounding the number of selectors to this depth. Any object name that exceeds this number of selectors is truncated, and then this new name is used. This breaks the uniqueness of object names: if two object names with k selectors are aliased, an infinite number of possible aliases are then represented.

Landi and Ryder show that their algorithm has only 3 cases of imprecision, other than the k-limiting discussed above. Importantly these 3 cases can all be measured, so the aliasing analysis can provide the approximate precision of the result. They also show empirically that their solution outperforms some common alternative methods.

3.7.3 Parametric Shape Analysis via 3-Valued Logic

In Parametric Shape Analysis via 3-Valued Logic [26], Sagiv et. al. attempt to construct a framework for describing shape invariants, which can be applied to any language and data structures. The important feature of this paper is allowing the shape to be parameterised, based on what data is desired, and hence a flexible approach can be maintained.

When designing a shape invariant, or abstract representation of a pointer structure, it is necessary to bound the size of the resulting graph. This means that there must be some nodes or pointers that correspond to more than one heap cell in the concrete program. When choosing what information to store, and what to discard, careful thought has to be taken as to the future use of the information.

Sagiv et. al. first set out a concrete representation of a data store, using boolean logic to specify properties about the heap cells. For example, a variable x would have a unary predicate such that $x(v)$ was only true when the variable x points to the heap cell v. Other predicates can be developed to indicate the use of selectors, for example $tail(u_1, u_2)$ would indicate that $u_1.tail = u_2$. These predicates can then be used to track the program state.

These predicates can represent the structure of a pointer system completely, but with this comes the problem of an unbounded number of predicates. To solve this problem, the authors turn to Kleene's 3-valued logic. Boolean logic can be modelled with values of 1 and 0, min instead of $\wedge$, and max instead of $\vee$. 3-valued logic introduces $\frac{1}{2}$ as a possible logical value. This new value is then used in some of the predicates to mean that the predicate may be either true or false. Certain nodes are then collapsed into one single node, and their predicates are merged.

The method for deciding which nodes to collapse uses the unary predicates. Initially the predicates consist of the variable names in the program, then all nodes are grouped by which unary predicates identify them. If two nodes have the same values for all the unary predicates then they are merged. In addition, a new unary predicate *sm* is added, tracking if this node is a merged node. This *sm* stands for same, and indicates the predicate that two nodes identified by this line are in fact the same node.

The important point about their method, is that it enables additional unary predicates to be added to the list, to distinguish more types of nodes. One example of a predicate used for this purpose is c_n, defined by

$$c_n(v) \stackrel{\text{def}}{=} n^+(v, v)$$

This is a test for cyclic paths, with this predicate being true if it is possible for the node to reach itself with any number of selector operations. This means that the end result of a cyclic list, and an acyclic list, are now distinct with respect to c_n. Even if part of the structure was acyclic, and part not, then this predicate would ensure they had different nodes in the graph.

Sagiv et. al. also define detailed semantics for an example programming language, including the effect of operations such as memory allocation via a `malloc()` statement. Update statements are given, and various properties about their resulting scheme are proven.

3.8 Summary

There are a variety of static analysis methods, which have been applied to real world programming problems in many ways, to obtain assurances about the code. Most standard static analysis tends to avoid the problems of pointers, or produce very suboptimal results in these cases. However, various methods exist for specifying possibly infinite data structures in a finite amount of space, by discarding certain information.

Chapter 4:

Design

This chapter deals with some design decisions made during the creation of the pasta analysis engine. Initial discussion of the Pasta language is included, along with totality.

4.1 Totality

In order to determine if a Pasta function is total, it is first necessary to identify what could cause the breaking of this property. Since the Pasta language is relatively simple, there are few classes of error that can occur.

4.1.1 Selectors

The Pasta term `x->y` is used to select the `y` field from the `x` variable. If the `x` variable does not have a `y` field then the program will crash at runtime. The Pasta language may have many subtypes with selectors called `y`. So the subtype of variable `x` does not have to be precisely known, but it must be within the set of subtypes having a `y` field.

4.1.2 Arithmetic

One subtle area which needs consideration is the arithmetic expressions, and in particular the result of undefined mathematical computations. The current version of Pasta does not permit any arithmetic operations to be performed, only numerical assignments and comparisons are permitted.

Should this situation change in the future, then it will be important to verify that all such operations are within the allowable bounds for the numeric type, or that any adverse effect (such as overflow being truncated, or wrapping round) is catered for, and the theoretical values updated appropriately.

In the current implementation of the Pasta language, the Pasta `int` type follows the conventions of the underlying system, being the Haskell `Int` type. This type is defined according to the Haskell standard[20], which states:

> The finite-precision integer type `Int` covers at least the range $[-2^{29}, 2^{29} - 1]$. ... The results of exceptional conditions (such as overflow or underflow) on the fixed-precision numeric types are undefined; an implementation may choose error ($\perp$, semantically), a truncated value, or a special value such as infinity, indefinite, etc.

In addition, when Pasta is converted to C, the `int` type is used. For C this is of an architecture dependant size, and overflow and underflow result in wrap around arithmetic.

For the moment this issue is ignored, as it is not possible to result in an arithmetic crash in the current version of Pasta, although this issue may require future work, should the specifications change.

4.1.3 Termination

Another way in which a Pasta function may not be total is if it could enter an infinite loop. In most programming languages there are two ways to do this, by use of an iterative statement, or using recursion. The Pasta language uses `while` to codify looping behaviour, and this must be checked. Recursion is disallowed in the language, as this conflicts with the graph analysis which it was designed for, so can be ignored.

4.1.4 Memory

The final class of error that can occur is memory exhaustion. If a Pasta program does not loop forever, then it must execute a finite number of steps. Since there is no Pasta statement that allows an infinite amount of memory to be allocated, it must be the case that only a finite amount of memory is required. While this is the case, it is impossible to predict the number of steps a Pasta program will take in general (this would be equivalent to solving the halting problem), and so it is impossible to set a bound on the amount of memory required.

This means that on any computer with a finite amount of memory, it is possible that a Pasta program will exhaust all the available memory, and hence result in a runtime error. This is not handled directly in Pasta, and it would be the underlying Haskell engine which would report this memory error. Since there is no bound given on the memory, and no memory semantics formalised by the Pasta language, I have chosen to ignore this issue, although it may present an interesting opportunity for further work.

4.2 Reduced Pasta

While the Pasta language is already a very small language compared to other languages such as C, the language still contains some redundancy. This section shows which statements can be removed from the Pasta language, and how they could be equivalently replaced. This reduced Pasta language actually has slightly more power than the original language, however since all programs will be derived from the original language, this issue can be avoided. In addition to following the semantics of the original language, the reduced language still has the same performance characteristics.

The following statements use many temporary variables, which I have given the name $\texttt{t}_n$, where n is an integer. These would be distinct for each production they are used in. At each stage various translations are made, with the intention that all generated statements at an early level are re-translated until they reach their most simple form.

For an example of the translation, see Appendix , which gives the expanded version of the singly linked list insertion example.

4.2.1 Assignment

For the purposes of reducing Pasta, I have concentrated on the assignment statement. This statement is the only one in Pasta that causes modification to the state of the heap, or any local variables, and hence is of critical importance. Pasta does not permit heap construction operations or copies in conditional expressions.

Parallel Assignment One statement in Pasta is parallel assignment, typified by a statement such as `a, x = b, y`. The semantics of parallel assignment under Pasta state that all the values are calculated before any of the results is assigned. The rules also say that no unstarred assignment may follow any starred one. From this it is easy to modify a Pasta statement to remove this parallel assignment, by storing all the intermediate results in temporary values first. This can be best illustrated with an example as in Figure 4.2.

Figure 4.2: Pasta assignment expansion

```
a, b, *c = x, y, *z;
-- becomes --
t1 = x; t2 = y; t3 = z;
a = t1; b = t2; *c = *t3;
```

As is obvious, each right hand side (RHS) expression is evaluated and stored in a temporary variable – the order of this evaluation is not important. Next these temporaries are assigned to the LHS's. In this case the order is important, and must be done from left to right to match the semantics of Pasta.

Assignment In a full Pasta assignment statement the LHS can be a variable followed by any number of selectors, and the RHS can be any Pasta expression. The assignment can also be between pointers or between integers, but both sides must be of the same type. Since the semantics of both types of assignment are completely different, it is useful to introduce the $\overset{\text{ptr}}{=}$ and $\overset{\text{int}}{=}$ assignment operators. These only operate on the appropriate type, while `=` can still be used for both. Since the Pasta language is statically typed, it is possible to convert any instance of `=` to a typed version, without difficulty.

In order to properly preserve the meanings, it is necessary to define the atomic statements given in Figure 4.3. In every case, `x` and `y` are variables, `field` is a selector and `new` is the name of a subtype.

Figure 4.3: Atomic assignment statements in Reduced Pasta

```
x = y;
x->field = y;
x = y->field;
*x ptr= *y;
x ptr= new();
x ptr= copy(y);
x int= 12;
```

It should be relatively easy to see how the conversion process can be carried out. In general, for an assignment `LHS = RHS`, first the `RHS` can be assigned into a temporary, and then assigned to the `LHS`. An example would be `x->head->tail = y->tail`, which gets converted to t_1 `= y->tail;` t_2 `= x->head;` t_2`->tail =` t_1`;`. The starred assignments are treated like ordinary assignments, with both the LHS and RHS being stored in temporary variables, before a final atomic starred assignment.

The one point that needs raising is the `x` $\overset{\text{ptr}}{=}$ `new();` statement, where `new` is the name of a subtype, for example `cons` or `nil`. While the heap cell creation statement in Pasta includes parameters representing all the fields, this version doesn't, as shown in [9]. A statement such as `x = cons(a, b);` would therefore be transformed to `x = cons(); x->tail = a; x->head = b;`. This is very useful, as otherwise a variable number of parameters need to be included in the heap cell construction statement, which complicates the definitions. A side effect is that it is possible in this reduced Pasta language to create a heap cell, and not initialise the fields within it, which is an undefined operation. This is the single area where the new definition has more power over the heap than the existing one, however if all construction statements are followed by assigning the fields (as would always be the case when converting from standard Pasta), this is not a problem.

4.2.2 Copying Expressions

The next expression that can be removed from the language is the `copy` expression. That this can be removed is mentioned in [25], with the remark "Providing copy as a primitive is convenient: it could be programmed as an operation for each data type."

The `copy` statement is dependant on the type signature at the start of the Pasta program. If a signature such as that given in is present, the equivalent procedure would be:

```
copy_list(ptr source, ptr destination) {
    if (source::nil)
        *destination = *nil();
    else
        *destination = *cons(source->head, source->tail);
}
```

Then the statement `x` $\overset{\text{ptr}}{=}$ `copy(y);` can be replaced with `x = nil(); copy_list(y, x);`. The initial

`x = nil();` statement is included merely to allocate a memory cell which can be overwritten later with the starred assignment.

4.2.3 Loop statements

With the `while` statement the main complexity arises from the guarding condition. In order to reduce this complexity, it is possible remove the expression out of the `while` statement, and replace it with a simpler expression. The format used to convert `while(condition) statement;` is shown in figure 4.4.

Figure 4.4: Reduced `while` statement

```
if (condition) {
    t1 = 1;
    while (t1 == 1) {
        statement;
        if (condition)
            t1 = 1;
        else
            t1 = 0;
    }
}
```

As can be seen, this reduces the complexity of the `while` condition to a comparison with 1. In fact, as this comparison is fixed, the `while` statement could be redefined to include this implicitly. The 1 and 0 are in fact representing boolean values, however Pasta has no literal boolean type, so integers are used instead.

An equivalent way of dealing with the `while` statement would be to introduce a `loop` statement that loops without a guard, and a `break` statement. This has some advantages, but resists mathematical analysis by distributing the iterative structure over more than one statement.

4.2.4 Conditional Statements

The complexity in the `if` statement stems mainly from the short circuit operators, `&&` and `||`. The problem here is that depending on the result of a previous expression, the next expression may not be executed and this conflicts with the natural mathematical approach. It is possible to remove these operators entirely, by expanding out the statements, for example `if (a ## b) x; else y;` is expanded as shown in Figures 4.5 and 4.6. The disadvantage of doing so is that sections of code are duplicated.

Figure 4.5: Expansion of `&&`

```
if (a)
    { if (b) x; else y; }
else
    y;
```

Figure 4.6: Expansion of `||`

```
if (a)
    x;
else
    { if (b) x; else y; }
```

The other reduction that can be performed is the removal of the `else` branch. This can be on the expression `if (a) x; else y;` as shown in Figure 4.7. There is however the question of whether removing the `else` statement actually reduces complexity, or increases it. While the `else` statement is an extra keyword, it makes the intent behind the above statement far more clear than the expanded version. Also, the `else` statement is relatively easy to define mathematically, and hence it has been retained.

Any final source of complexity now rests in the conditions, which are executed to completion at every `if` statement. The two conditions that are available are subtype checking `x::cons` and relational operators `x > y`. In both these cases, the expressions `x` and `y` can be any variable followed by a chain of selectors, and additionally for the relation operators can be a literal number. In both these cases, `x` and `y` can be replaced by single variables, without loss of power. In the case where a selector is involved, the atomic statements discussed earlier can be used. For the relational operators, where a literal is used, this can first be assigned

to a variable. As an example, translating `if (x->head > 4) statement;` could be transformed as shown in Figure 4.8.

4.2.5 Subroutine calls

The other form of structure in Pasta is subroutine calls. The Pasta language forbids recursion, both direct and indirect, and hence all subroutines can be expanded inline without resulting in an infinite amount of code. This does result in a large number of code statements, but removes some complexity.

4.2.6 Scope renaming

Another feature of Pasta that can cause problems from a mathematical analysis point of view is scope renaming. In Pasta, when a variable name is encountered the program looks for the variable which such a name in the innermost enclosing scope. This presents a problem because it means that two distinct variables may be referred to by the same name. This can be solved quite easily by appending the depth of the variable onto the name, and resolving all names at this point.

Figure 4.7: Removal of `else`

```
t1 = 0; if (a) {
    t1 = 1;
    x;
} if (t1 == 0)
    y;
```

Figure 4.8: Simplification of comparison

```
if (a)
    x;
else
    { if (b) x; else y; }
```

4.2.7 Typing

A final point to mention on this reduced Pasta language, is that all the declaration statements have been removed. For example, in standard Pasta `ptr p` is used to declare a variable called `p` which is a pointer. In actual fact, the declarations are redundant from the meaning of the language, other than being additional assertions the programmer can use to state their intent more clearly. As a result, these definitions can be removed, and all variables can be considered to be in the outermost scope, with their type being inferred by their use.

The resulting Reduced Pasta language removes some of the complexity of the Pasta language, but at the cost of a vastly larger number of statements and temporary variables. This new language is suitable for analysis, but is not intended to be used by programmers.

4.3 Completeness of Reduced Pasta

Now reduced Pasta language has been constructed, it is possible to define what totality means in terms of the Pasta language in much more concrete terms. The one issue that is not touched upon is the typing, as this is an easy problem to solve, and has already been fully covered by the initial Pasta compiler.

4.3.1 Selectors

There are only two statements in the reduced version of Pasta that involved navigating down a selector, these are `y = x->field` and `x->field = y`. The expression `x->field` is only valid if the subtype of the heap cell pointed to by `x` has a selector named `field`. Another way of phrasing this would be that the subtype of `x` must be one of the set that contains `field` as a selector.

4.3.2 Termination

Since Reduced Pasta has no procedure calls, the only statements that can repeat are within the `while` construct. Of course, it is impossible to prove termination in the general case, because the Reduced Pasta language Turing complete. However, there are several mechanisms for proving that a large subset of a particular language will indeed terminate.

First it is necessary to investigate the statement `while(`k `== 1)` x`;`, to determine what termination means in this context. For a `while` loop to terminate, it must eventually be the case that the condition does not hold, and here this means that $k \neq 1$. If k^n is used to denote the value of k after n executions of the statement x, the condition of termination can then be given as:

$$\exists i \in \mathbb{Z} \bullet i \geq 0 \land k^i \neq 1$$

While this is the general case, there are several restricted cases where termination can be proved, and these are covered below.

Bounded State Spaces In a deterministic language, the action of the code at any point is determined by the state of the program. In the context of loops, this means that whether the loop is executed again will depend only on the state of the program at that particular point. If a loop is in a particular state, and after n executions of the loop body, returns to the same state then a periodic fixed point is established, and it is clear that the loop will never terminate.

While this is a method for detecting non-termination, it cannot be applied so easily for proving the reverse. The reason for this is that the length of the period n is unbounded, and hence it is never possible to say that sufficient values of n have been checked. However, if the number of distinct states is bounded, then the maximum value of n is equal to this number.

One example of a routine with a fixed state space is shown in Figure 4.9, where the state space is bounded to 100 values for i, provided i is defined to be an integer. At runtime, if this loop is executed 101 or more times, then it will never terminate. During static analysis, the value of i before the loop is unlikely to be known, but it is possible that i could be narrowed down to a finite number of states. Each of these states can be tested on the loop, for the maximum number of times. This would then either prove termination, or give a specific state in which the program does not terminate.

Figure 4.9: Example of a fixed state space

while$(i \neq 0)$ `{`
    $i = (i \times 3)^2 \bmod 100$`;`
`}`

In Pasta the state at any point consists of the pointers between heap cells and from variables, and any literal integers present in the program. Since it is not possible to create any new literal integers (due to the lack of arithmetic), it is clear that if the number of integers remains fixed, then the integer state is limited to the exchanging of values, and hence is bounded. In a similar way, if the number of heap cells and pointers remains fixed, and all pointers point at a valid heap cell, then only pointer rearrangement is possible, which is also bounded.

The one thing that can increase state space is the heap cell creation statement, such as `new()`. If this statement occurs within a loop, then there is no easy way to determine an upper bound on the state space. This does however mean that if the `new()` statement does not occur inside a loop, then it is possible to test if the loop terminates, provided the state before the loop was finite.

Putting these observations together, it is possible to see that if a Pasta program does not contain any unbounded state loops, then the amount of state in the program is finite. This means that if the program does not contain any `new()` statements within `while` statements, then the program can be proved to either

terminate or not terminate. A large number of programs do not contain heap allocation in loops, including linked lists, queues, trees and some sort algorithms. However, while manipulating linked lists can be done in this way, it is quite possible that they will be created in a loop statement.

Another problem with this method is its feasibility. While the number of states is bounded, it is by no means a small number, and in fact for simple structures is likely to be significantly out of a computable range. As an example, if there are n pointer variables, and m cells in the heap, the number of program states would be at least m^n.

Since it possible to determine termination information for this restricted subset it is clear then that this is not Turing complete. In fact, the expressive power of this language is equivalent to that of a linear-bounded Turing machine. While this result is useful theoretically, its restrictions are too great to use as the basis of a termination proof.

Fixed Loop Iterations One common program pattern is to iterate over a predetermined range, for example the `for` statement in Ada requires the number of iterations to be specified before starting the loop. This means that the `for` statement is guaranteed to terminate.

While this is a useful construct, it is unlikely to be of great use in Pasta. The most common use of this statement is for iterating through arrays, incrementing an index each iteration. In Pasta there are neither arrays, nor arithmetic, so this statement would not be as useful as for Ada.

Traversal down a finite path One form of fixed loop iteration in a pointer programming language is traversal down a finite path. If a loop proceeds down a finite path, then the number of executions of the loop is bounded by the length of the path. While it is relatively easy to prove that a loop proceeds down a path, the difficult thing is determining if this path is finite or not.

For example, it is clear that the following loop proceeds down the tail pointers of the variable `x`:

```
ptr x = ...;
while(x::cons)
    x = x->tail;
```

And on a typical linked list, this program would terminate, for example if `x` was:

However, it is not necessary that the path down `x`, following `tail` selectors is finite. Consider the following program structure for `x`:

For traversal down a finite path to prove terminate, it must be the case that the loop travels down a path by at least one node during every execution, and that any other operations in the loop body do not affect the structure of the path. This information can be determined statically in a range of cases. It is possible that more than one path traversal will occur during a loop. This means that the safety condition for the `while` statement is that there exists a finite path out of those paths that are traversed.

The final question is what will happen if the end is reached? In the above example the `x::cons` checks that the end has been reached. However, if that expression was replaced with a tautologous expression, and the path was indeed finite, then a crash would occur. The finite path traversal argument does indeed work as proof for termination, but does not specify whether that termination is safe or unsafe. Fortunately this approach can be combined with the safety analysis of selectors, and safe termination can then be proved.

4.4 Full or Partial Analysis

A final question that needs to be addressed is the type of analysis being performed. One type of analysis is full analysis, where an entire program is taken as a unit and analysed. This is in contrast to partial analysis, where only a subroutine within a program is analysed. While the two types can be implemented in very similar ways, as a procedure taking either no unknown variables, or some unknowns, they can have very different design goals.

One particular instance where the difference is pronounced is the reaction to a failure. If an individual subroutine does not pass the analysis, then this is something to examine, but it may be that the routine is being called without its preconditions satisfied. On the other hand, if a program in its totality does not pass, then this indicates a clear failure by either the program (performing incorrectly), or the static analysis (failing to correctly interpret the program).

For the subsequent chapters, I have tried to cover both types of analysis, to produce a tool that is suited to all cases. Where a design decision is particularly influenced by one tasks or the other, this has been mentioned.

4.5 Summary

In this chapter a Reduced Pasta has been presented, and its relationship to standard Pasta has be shown. The totality of Reduced Pasta has been examined, including both selector safety and various termination arguments.

Chapter 5:

Forward Analysis

This chapter takes the Reduced Pasta language as set out in the previous chapter, along with its requirements for totality, and expands it into an analysis program that could be implemented. The focus of this chapter is a forward analyser, however much of the discussion is generally applicable to other types of analysis.

Forward analysis works by tracking all possible states the program could be in at any point, and then executing statements to transform this abstract state. From the abstract state, it is then possible to determine if the safety preconditions for each statement hold. One of the biggest challenge when designing a forward analyser is choosing what information to discard, in order to keep the information at a bounded size.

5.1 Notation

In order to accurately represent the effect of statements, the notation $[\![x]\!](Q) = Q'$ is used to mean that if the system is in state Q before statement x, then afterwards it will be in state Q'.

To check if a precondition is true in a given state, the predicate $pre(Q, r)$ is used, where Q is a state, and r is a condition. The pre predicate returns true only if the condition r definitely holds in state Q.

In order to reduce two states into one, the operator $\odot$ is used. The $\odot$ operator can be thought of as merging two knowledge states when only on of the states is correct, and can be defined as:

$$\forall r \bullet pre(Q_1 \odot Q_2, r) \Rightarrow pre(Q_1, r) \wedge pre(Q_2, r)$$

The $\sqsubseteq$ operator is used to mean that everything in the LHS state can be inferred from the RHS. This can be defined as:

$$(Q_1 \sqsubseteq Q_2) \Leftrightarrow (\forall r \bullet pre(Q_2, r) \Rightarrow pre(Q_1, r))$$

The function $add(Q, r)$ is used to add the condition r into the state Q. This can be defined as:

$$\forall r \bullet add(Q, r) = Q' \Rightarrow Q \sqsubseteq Q' \wedge pre(Q', r)$$

5.2 Compound Statements

The first thing that needs considering is how the state will be mapped over the various statements that are encountered, and appropriately updated. Before defining the effect of the individual assignment statements, it is first useful to consider larger groups of statements, assuming these atomic statements have been defined. The three groups that are focused on are `if` statements, `while` statements and sequences of statements.

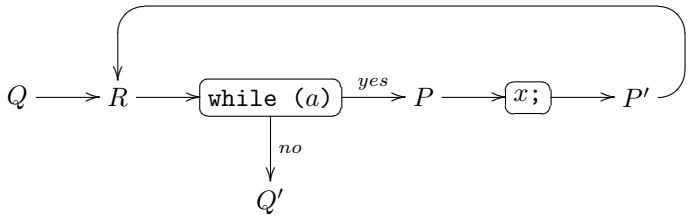
5.2.1 Sequence

The most simple compound statement in Pasta is the sequence, for example $x; y$, the result of executing x and then y. This can be defined quite simply as:

$$[\![x; y]\!](Q) = [\![y]\!]([\![x]\!](Q))$$

5.2.2 Condition statements

Figure 5.10: General while statement



When analysing an `if` statement, it may sometimes be possible to determine the outcome of the condition, and hence exclude one branch from the analysis. While at first appearance, this may suggest an area of the code which cannot be executed, this is not the case. During a loop, it is quite possible that the first iteration will invoke one branch of the `if` statement, while all subsequent iterations will invoke the other one.

In the normal case, where either branch may be executed, the test performed will yield some information about the state of the program. If the `else` branch is taken, then the complement of the test must be true. After both branches have been executed, there will necessarily be two distinct states, and these need to be merged into one to continue the next statement. An `if` statement can therefore be analysed with:

$$[\![\texttt{if (}a\texttt{) } x\texttt{; else } y\texttt{;}]\!](Q) = \begin{cases} [\![x]\!](Q) & \text{where } a \text{ is true} \\ [\![y]\!](Q) & \text{where } a \text{ is false} \\ [\![x]\!](add(Q,a)) \odot [\![y]\!](add(Q,\neg a)) & \text{otherwise} \end{cases}$$

5.2.3 Loop statements

The `while` statement is far harder to model than the other statements, due to its repetitive nature. For the following initial analysis, it is assumed that the loop terminates – detecting if this is not the case is discussed later.

A loop can be generalised as in Figure 5.10. In this diagram, Q is the state before analysing the loop, R is the state just before the guard and P is the state in which the preconditions of x must be satisfied. The state at each point can then be defined by:

$$\begin{aligned} R &= Q \odot P' \\ P &= add(R, a) \\ P' &= [\![x]\!](P) \\ Q' &= add(R, \neg)a \end{aligned}$$

In order to calculate the knowledge after an unbounded number of executions, it is necessary to establish a fixed point to the state. A fixed point state at R would be one where additional executions did not effect it, defined as:

$$(R \sqsubseteq [\![x]\!](add(R,a)) \wedge (R \sqsubseteq Q)$$

The simplest value for R would be no knowledge, and this is indeed a fixed point, although hopefully this would be a worst case. From this fixed point R, it is easy to determine P and Q', using the equations given above.

One way of finding a fixed point is to define R^n as the state just before the $(n+1)$th execution of the `while` loop, with R^0 being the initial Q parameter. It is then possible to define R^{n+1} as:

$$R^{n+1} = R^n \odot [\![x]\!](R^n \wedge a)$$

If the $\odot$ operator is defined in such a way that $(Q' = Q_1 \odot Q_2) \Rightarrow (Q' \sqsubseteq Q_1) \wedge (Q' \sqsubseteq Q_2)$, and that the amount of knowledge in a state can be mapped to a positive integer, then this terminates. The intuitive understanding of this would be that whenever two states are joined, no new information is created. By performing a $\odot$ with the R^n state to generate the R^{n+1}, this means that $R^{n+1} \sqsubseteq R^n$. If they are equal, then a fixed point has been achieved. If they are not equal, then the amount of knowledge must have been reduced by some amount. If the reduction proceeds repeatedly, then eventually the state containing no knowledge will be reached, at which point a fixed point will have been achieved. This can be thought of as the R^∞ state.

An alternative mechanism can also be developed, which retains more information until the end of the computation, albeit requiring a larger number of computations. An alternative (and possibly more natural) definition could then be given for R^{n+1} omitting the $\odot$ with R^n.

$$R^{n+1} = [\![x]\!](R^n \wedge \mathtt{a})$$

It is no longer the case that $R^{n+1} \sqsubseteq R^n$, meaning that knowledge is not needlessly destroyed. R^∞ can still be found by applying the $\odot$ operator to the states, until a fixed point is found:

$$R^\infty = R^0 \odot R^1 \odot R^2 \odot \cdots$$

This new version of R^∞ can be used to generate P, but it does not need to be used in the computation of Q'. Since all the intermediate results are already known, U^n can be defined as the state after the loop, when n iterations have been performed. U^n can be defined as:

$$U^n = add(R^n, \neg a)$$

It is then possible to define U^∞ in the same way as was done for R. U^∞ is then equal to Q'. This method is at least as precise as the previous method, while offering the opportunity for more precision.

The trade off in choosing the method for determining Q' is one of the superior speed of method 1, versus the possible superior accuracy of method 2. To actually determine the size of the difference in these categories would require experimentation and empirical results.

5.3 Atomic Pointer Assignment

The effect of an atomic pointer assignment statement needs to be modelled, with a view to deducing if the require safety preconditions for statements are true. The information that is used in preconditions, that is most affected by assignment, is the safety of selectors. This can be done by tracking the subtypes of heap cells.

The easiest atomic statement to deal with is the `x` $\overset{\text{ptr}}{=}$ `new()`, which sets `x` to point at a new heap cell, of subtype `new`. The `::` operator, as used in the `if` statement, also provides information on subtypes. This condition enables the subtype of a heap cell to be known exactly, or in the case where the condition fails, one possibility to be excluded from the possible types.

None of the other atomic statements give knowledge about the type of a heap cell when viewed in isolation. However if the state before contains information about the subtype of a cell, some information may be determined. The most simple statement, `x = y`, assigns `x` to point to the same heap cell as `y`. If the subtype

of `y` is known before, then afterwards the subtype of `x` will also be known. This is an example of adding knowledge to the state. But any information stored about `x` before `x = y` will become invalid, and needs to be removed. This includes both `x`, and any selectors from it, for example the subtype of `x->tail` is now no longer known. On the other hand, all information about the selectors of `y` now also applies to `x`, for example `x = y` means `x->tail` and `y->tail` are both the same heap cell (assuming `y` has a `tail` selector).

The statement `x = y` also creates an aliasing. The predicate $alias(x, y)$ is used to mean that the variables x and y are aliased – this relation is both symmetric and transitive. An alternative model of the heap is as a set of *alias* relations, with some additional information indicating which heap cells are of known type. Then all the assignment statements either mutate the aliasing relations, or the type information. Tracking aliases ensures that if two nodes point to the same heap cell, and that heap cell is mutated (for example using starred assignment), then this can be tracked.

The starred assignment statement is also very hard to model. The statement `*x = *y` mutates the heap cell that `x` pointers to, and any variables that are aliased to `x` are also mutated. This means that tracking aliasing information is clearly a very useful operation.

Tracking complete alias information in a program is likely to be impossible, since the $\odot$ operator may mean that the *alias* predicate cannot be determined exactly. Also, to track the aliasing in an unbounded linked list would require an unbounded amount of *alias* relations, which is clearly not feasible. As a result it is likely that the *alias* predicate will not return boolean values, but rather use Kleene's 3-valued logic. In this new definition of *alias*, 0 represents that the two pointers are not aliased, $\frac{1}{2}$ represents that they may be, and 1 that they definitely are.

In most static-analysis work, the program is concerned mainly with the local variables and their values, which are finite in size. However, with Pasta, the analysis is mainly concentrating on data structures on the heap, which may have other data structures referenced within them. This means that instead of being finite in scope, there is potentially an infinite amount of state. This can be modelled in two contrasting ways, each giving different complexities to different operations.

5.3.1 Infinite Domains

The first view of the problem is to see a finite number of variables, but each with an infinite range of values it could take. In this representation child nodes would be stored as children of the parent. This has advantages for traversing an entire variable, and is possibly easier to perform techniques such as shape analysis. In this view the statement `x = cons(12, nil())` would have one variable `x`, with the value `cons(12, nil())`.

5.3.2 Infinite Variables

A totally different view of the variables is to consider each variable as either a subtype or an integer, with the children of the node as separate variables. This results in an infinite number of variables, but with a finite domain for each variable. This makes tracking of the aliasing information easier, and holding the integer values used in comparisons becomes slightly more accessible as well. In this view `x = cons(12, nil())` would create the variables `x = cons()`, `x->head = 12` and `x->tail = nil()`.

The first intuitive observation on these two options is that the infinite domains representation more closely models Pasta, whereas the infinite variables representation is closer to Reduced Pasta. Despite this, both models can be used when working from either dialect of Pasta.

Infinite variable works much better for aliasing, and allows variables to alias into the middle of other heap structures. Using this representation it is also the case that if $field$ exists in both x and y, and is a `ptr`

type, then:

$$alias(x \rightarrow field, y \rightarrow field) \geq alias(x, y)$$

This relation uses the fact that aliasing can be 0, $\frac{1}{2}$ or 1. If the initial variables x and y are aliased, then their common selectors must be, but even if they are distinct, their selectors could still be aliased.

It is useful to define $\varphi(x)$ as the set of possible subtypes of the heap cell pointed at by x. If nothing is known about x then $\varphi(x)$ would be the set of all subtypes. When combined with $alias$, it can be seen that if $alias(x, y) = 1$ then the possible subtypes for x are actually $\varphi(x) \cap \varphi(y)$.

With these primitives, it is possible to define more precisely the effects of the atomic statements.

5.3.3 Simple variable assignment

The most simple assignment statement, $x = y$, has the effect of aliasing x and y, making $alias(x, y) = 1$. Also the update destroys all information about x, and makes $\varphi'(x) = \varphi(y)$. In addition, $\forall z \bullet alias(x, z) = alias(y, z)$. Another way of stating this would be that all existing alias relations with x are destroyed, and instead those related to y are used instead.

5.3.4 Assignment from field values

The next statement, $x = y \rightarrow field$, also creates the aliasing $alias(x, y \rightarrow field) = 1$. Also, in a very similar manner to $x = y$, it makes $\varphi'(x \rightarrow field) = \varphi(y \rightarrow field)$, and destroys existing alias information.

5.3.5 Assignment to field values

The statement $x \rightarrow field = y$ adds more complexity to the situation, by affecting more variables than just x and y. The additional $alias(x \rightarrow field, y) = 1$ relation is an obvious effect, and also as before $\varphi'(x \rightarrow field) = \varphi(y)$. The interesting cases come from examining the effect of this statement on a different variable, z. If $alias(x, z) = 0$ then this statement does not affect $\varphi(z)$. If $alias(x, z) = 1$ then afterwards $alias(z \rightarrow field, y) = 1$ is the case.

The final case, where $alias(x, z) = \frac{1}{2}$ is the most interesting. In this case, $\varphi'(z \rightarrow field) = \varphi(z \rightarrow field) \cup \varphi(y)$. Also $z \rightarrow field \rightarrow field$ may be aliased to $y \rightarrow field$, meaning that $\varphi'(z \rightarrow field \rightarrow field) = \varphi(y \rightarrow field) \cup \varphi(z \rightarrow field \rightarrow field)$. This assumes that the $field$ selector exists for both $z \rightarrow field$ and y, otherwise all information about $z \rightarrow field$'s children may need to be destroyed. In this case, a small amount of imprecision in the aliasing may lead to a large amount of imprecision in the resulting state.

5.3.6 Starred Assignment

The case of a starred assignment $*x = *y$ is also complex, and also requires modification to variables not involved in the statement. The major difference from all other statements covered so far is that after this statement $alias(x, y) = 1$ is not guaranteed to hold, and in fact retains its original value. If this expression does hold before the evaluation of the assignment, then the assignment has no effect on the state of the heap.

After the assignment, $\varphi'(x) = \varphi(y)$ and all the selectors that x possesses are now aliased to those of y. For example, if y has a selector named $field$, afterwards $alias(x \rightarrow field, y \rightarrow field) = 1$ will be true.

For all other variables, such as z, if $alias(x, z) = 0$ then $\varphi(z)$ does not change. If $alias(x, z) = 1$, then $\varphi'(z) = \varphi'(x) = \varphi(y)$. Again the interesting case is where $alias(x, z) = \frac{1}{2}$, in which case $\varphi'(z) = \varphi(z) \cup \varphi(y)$. In a similar manner to $x \rightarrow field = y$, information about the children of z may need to be destroyed.

5.4 Atomic Integer Statements

To add extra power to the static analysis, it is useful to track the value of integer variables. However, while many static analysis programs focus on integers, the purpose of this static analyser is to focus on the heap data structures. As a result the numeric analysis can be limited. No preconditions for safety require any numeric values to be known, instead the use of numbers is to precisely determine which branch of an `if` statement will be taken.

5.4.1 Range analysis

One approach is to track the possible ranges of a value, and use that to determine the result of computations. This is an approach that has been used extensively by other analysis tools. When a comparison is made, then the ranges can be used to check if the result of the comparison can be determined. The $\odot$ operation can be easily implemented, and mathematical operations on integers can be performed on ranges with reasonable precision.

One specific problem with range analysis comes when an assertion is made about the relationship between two unknown variables, e.g. $a < b$ where the values of neither a nor b are known. In this case, the simple range analysis techniques do not increase the available knowledge. Looking through samples of Pasta programs, this seems to be by far the most common situation, and hence range analysis is inappropriate for the Pasta programming language. Also the major focus of range analysis is the effect of mathematical operators, which do not exist in Pasta.

5.4.2 Constraint analysis

An alternative to tracking the ranges of a particular variable is to accumulate constraints about all variables. This delays processing any of the information until information is requested about the values of the variables. This is the optimum solution, as no information is discarded at any stage, however constraint solving is an NP-Complete problem [8]. Since the focus in Pasta not on numeric work, this seems hard to justify as an expense.

5.4.3 Trichotomy

One particular example of reduced constraints uses the law of Trichotomy, which can be expressed as:

> For arbitrary real numbers a and b, exactly one of the relations $a < b$, $a = b$, $a > b$ holds

It should be mentioned that while this law applies to Mathematics, it does not apply to computer programs in general, where reserved values such as `NaN`, the "Not a Number" constant, are in effect. With the Pasta language, there is no such problem of invalid numbers, and hence this law always holds.

For any two integer variables, their known relationship can be stored as an element of the powerset of $\{<, =, >\}$. If no information is known then the set $\{<, =, >\}$ represents the knowledge available. While it may appear non-intuitive, it is possible for the static analysis to determine that the relationship between two variables is $\emptyset$; that is there is no possible relationship between them. This can be interpreted as meaning the program has entered an impossible state, for example `if (x > y && x < y) statement;` will never executed `statement`.

5.4.4 Transitive Trichotomy

From the basic trichotomy information it is possible to return information that has been already stored. There are various laws about the transitive nature of relations, for example:

$$a > b \wedge b > c \Rightarrow a > c$$

A number of these laws exist, and these can be used to increase the amount of information in the inferred. One method for performing this is when a new relation is added to the set. If a relation between a and b is added, then the new relation between a and b will be the intersection of the new relation and the existing one. If this changes the existing relation then compare all relations where either a or b is a member, and see if the relation set can be reduced. If it can, then replace the relation, and add it to a queue. Keep taking a relation off the queue, performing transitive updates and keep going until there are no more relations on the queue. At this point the transitive closure of the relations will have been found.

5.4.5 Numerical Assignment

In Pasta, while there are not any mathematical operations, there are assignment operations, and these add information to the state. Where numbers are given explicitly, e.g. `int a = 100;` this does not fit easily into the store of relations between numbers. One approach is to store the numerical values of all the known integers, and then when another known value is added, insert relationships between all existing values and the new one into the relation list.

5.5 Aliasing

As has been shown earlier, in Pasta the effect of a statement varies greatly depending on what is aliased to what, i.e. the *alias* predicate. The power of the aliasing determination is likely to play a great role in the power of the resulting analysis. With the Pasta programming language, when focusing on a subroutine in isolation, it is not possible to determine full aliasing information. One reason for this is that the data structure given as an argument may alias other things, either within the data structure it points at, or other arguments, and this cannot be determined.

In order to track the aliasing information at each stage I came up with two distinct representations, each with advantages and disadvantages. In both cases, if a variable does not have any aliasing information associated with it then it is in the Unknown set, otherwise it is in the Known set.

5.5.1 No Aliasing Information

Of course, a simple implementation could ignore aliasing entirely, and this would be equivalent to

$$\forall x, y \bullet alias(x, y) = \tfrac{1}{2}$$

However, without any more resources, certain cases can be refined:

$$\forall x \bullet alias(x, x) = 1$$

5.5.2 May be aliased

The first representation I considered maintains two sets of information, the Def set representing those variables which are aliased, and the May set representing those which may be aliased. Both Def and May are sets of sets which contain variables. With these sets the alias information can be determined using the following equations.

$$\forall A, x, y \bullet A \in \text{Def} \wedge x \in A \wedge y \in A \Rightarrow alias(x, y)$$
$$\forall A, x, y \bullet A \in \text{May} \wedge x \in \text{Known} \wedge y \in A \wedge x \notin A \Rightarrow \neg alias(x, y)$$

Def sets can be thought of as a partitioning of all the Known variables. If two variables are in the same partition then are definitely aliased.

The May sets contain variables that may be aliased. If two known variables are not in the same May set then they are not aliased.

Any variable that is in the Unknown set may be aliased with any element in the Known set at all, and this representation returns Unknown for all such alias requests.

This representation naturally models the idea of aliasing, but has a disadvantage of always assuming the worst with respect to Unknown variables. It is also simple in this model to tell what variables that are known may be aliased to each other.

5.5.3 Uniquely aliased

An alternative model is to store those things that are uniquely aliased in the Uni set, and those that are non-uniquely aliased in the Shr set, both being sets of sets of variables. If two things are uniquely aliased that means that no other variables point at that memory location. If two elements are non-uniquely aliased, then other variables may also point at that location.

$$\forall A, x, y \bullet A \in \text{Uni} \land x \in A \Rightarrow (y \in A \Leftrightarrow alias(x, y))$$
$$\forall A, x, y \bullet A \in \text{Shr} \land x \in A \land y \in A \Rightarrow alias(x, y)$$

The advantage of this approach over the alternative is that information is now known about the Unknown set, as a variable in the Unknown set may not be aliased to a variable contained anywhere within a Uni set. Another large factor in favour of this approach is that it is easier to generate the required knowledge from the program constructs.

While at first glance this method seems to be better suited to the Pasta language, experimentation would be required to determine this for certain. However, both approaches can exist simultaneously, and the resulting *alias* function can then choose whichever method results in a higher information content, where $\frac{1}{2}$ has lower information than either 0 or 1.

5.6 The $\odot$ operation

One of the major operations in this static analysis method involves the $\odot$ operator. While an actual $\odot$ operator is not detailed here, this section focuses on the ideas surrounding it. The $\odot$ operator is defined above as:

$$\forall r \bullet pre(Q_1 \odot Q_2, r) \Rightarrow pre(Q_1, r) \land pre(Q_2, r)$$

The $\Rightarrow$ is particularly important in this case, as it means the $\odot$ operator may be lossy. An example of a valid (albeit very pessimistic) $\odot$ would also result in a Q' such that $\forall r \bullet \neg pre(Q', r)$. This would be equivalent to failing every safety precondition.

The implication may appear to mean that information would have to be discarded at each stage, but this is not the case. One possible construction of $\odot$ on an `if (a) b; else c;` statement would be just that information that is true at the end, with all intermediate results being destroyed. An alternative, and more knowledge preserving $\odot$, would be to store that if `a` is true one state is the case, and otherwise another is. If consequently it turns out that `a` is true, then one portion of the knowledge could then be rejected, without having diluted the valid knowledge.

The primary disadvantage of this definition would be that the termination of the `while` statement would no longer be guaranteed. One way to remove this problem would be to define more than one $\odot$ operator, one

which is lossy, and one which is not. Then by using the lossy one where other arguments require it, and the non-lossy one elsewhere, more knowledge could be preserved.

5.7 Acyclic Paths

Up until this point the information has been used mainly to determine the appropriate subtype of any heap cell. While the subtype proves the selector operation safe, it does not help with the termination argument. The termination argument relies on finding a variable in a `while` statement with forward motion down an acyclic path.

A variable has forward motion if down every possible route through the body of the `while` statement, a variable is moved from one position in the path to a subsequent one. Checking that the path is traversed down every branch means that if the update occurs on one branch of an `if` statement, then it must occur on the other branch as well for this to be a valid path. The requirement that the variable is moved down means that other operations in the `while` body do not interfere with the variable in question. This in particular excludes starred assignment and $x \rightarrow field = y$ statements, where x may be aliased to the path variable.

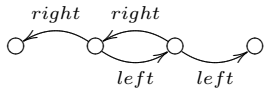
The second problem, determining acyclicity, is far harder. Acyclicity could be determined by aliasing information, although this would require all aliases to be determined as one single $\frac{1}{2}$ result could break this argument. Another way to check for this property is by using shape analysis techniques, as described in [26]. Both these techniques are only useful when the input at the start of the program can be determined, and not for validating a procedure on its own with undefined inputs.

When the input at the start of the procedure is not known, it becomes impossible to automatically derive acyclic arguments, and so for a large number of cases termination cannot be determined. Unfortunately this is likely to be a very common scenario, and so an alternative mechanism is required. Any manual solution must involve the programmer adding annotations to the program, which is unfortunate, and a goal should be to minimise the complexity of these annotations.

The definition I have chosen to add to Pasta is the `acyclic` keyword. The format is `acyclic(`$selector_1$`, `$selector_2$`, ..., `$selector_n$`)`, where the elements in $selector_i$ form a set. This says that from any heap cell, following any number of selectors in the set, the original heap cell will not be reached. When restricted to a set of selectors, there are no cycles in the data structure. This statement is applied at the signature level, so the initial example given in would become:

```
list acyclic(tail) {
   nil();
   cons(int head, ptr tail);
}
```

It should be noted that the statement `acyclic(left, right)` is a much stronger assertion than `acyclic(left) acyclic(right)`. For example the following heap state is allowed by the latter, but not the former:



Using the `acyclic` assertions provided, the static analyser can determine which paths that are traversed are acyclic with ease. Having an annotation of acyclicity does not guarantee this property will be true, and

would require the programmer to check this manually. This would mean that the program could only be proved to the same degree as the `acyclic` property.

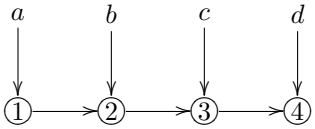
One solution to this problem is to assume the `acyclic` property holds before the execution of the current routine. Then the analysis can be responsible for ensuring that if the precondition of acyclicity is true, then afterwards the structure must also be acyclic. Viewed this way, the acyclic property serves mainly as a hint to the analyser that this is a good choice to use for checking.

If a program has n pointer selection fields, then there are 2^n possible acyclic statements that could be written. Of these, one would be the trivial `ayclic()` statement, which is always true. This leaves $2^n - 1$ possible statements that could be written. The number n is likely to be relatively small for most programs, for example in the linked list example this number would be 1, and even for more complex data structures such as doubly threaded trees and 234 trees, this number does not exceed 4. This would mean that $2^4 - 1 = 15$ possible acyclic statements can be generated. This number is small enough so that checking each statement using the analysis is feasible, and then after this has been done, those that are always true can be retained.

The disadvantage of automatically generating `acyclic` statements is that the programmer no longer is responsible for determining the properties of the program. Properties that are not violated, but equally are not intrinsic properties of the structure, may be used. The other problem is when analysing small sections of the program in isolation, it is impossible to see what properties would be violated in other sections of the program. For this reason, a tool to generate `acyclic` properties would be useful, but relying on given assertions is probably the right course to explore.

Once the acyclic properties have been determined, they then have to be checked. These properties are similar to invariants in an object-orientated program, and from this various approaches can be garnered.

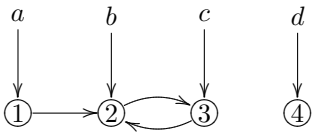
Some languages require the invariants to hold at all points, while others only require that the invariants hold at interprocedural boundaries, not during the body of a procedure. An example of a case where an acyclic data structure may become momentarily non-acyclic would be swapping two adjacent elements in a linked list. For example, consider the following structure:



Suppose elements 2 and 3 must be swapped. One possible piece of code to do this would be:

```
c->tail = b; b->tail = d; a->tail = c;
```

This code does give the correct result, but consider the situation after the first statement has been executed:



In this code sample, the acyclic constraint has been temporarily violated. The question then becomes whether to disallow this sequence of statements in that order, or require that the ordering of code be changed, or temporaries be created, so that always the acyclic property is preserved. It is likely that in most cases an equivalent sequence of code can be derived that preserves the property at all times. This could generally be

achieved by pointing all the fields at a new temporary heap object, and then repointing them to their final destination afterwards. An example of how to rewrite the above code would be:

```
a->tail = c; b->tail = d; c->tail = b;
```

This sequence does preserve the property at every step, and produces identical end results. I have decided to disallow the breaking of this constraint at any point, partly because of the ease of rewriting in the occasional case where this property is violated. Another reason is that if the acyclic constraint is violated with a loop statement then the termination argument would be broken. For this purpose procedure boundaries are not sufficient.

If an acyclic property must be true at all times, then atomic pointer statement must preserve it. To achieve this, it is necessary to analyse each of the statements. The $x = y$ statement, and the $x = y \rightarrow field$ statement are both fine, as they only effect the aliasing of the variables, and not of the heap cells.

The statement $x \rightarrow tail = y$ is harder to prove. The most obvious case is that y must not point at x through any cycle of *tail* pointers. This is, the set of nodes which point to x through any number of *tail* selectors, must be disjoint from those to which y points via *tail*. This also takes care of variables that alias to the same heap cell as x, as they must point at x through *tail* selectors.

One restricted instance where acyclicity can be proven is when the value of y has just been created using a $new()$ statement, as nothing else can yet point at this new cell. Another case is when the y heap cell is of a subtype which does not have *tail* selector. Finally, another case is when this statement results from a $x \rightarrow tail = x \rightarrow tail \rightarrow tail$ statement, the effect being to delete a node from the list.

The last remaining statement, $*x = *y$ is even trickier. To preserve acyclicity, for every variable z, where $alias(z \rightarrow tail, x) > 0$, the conditions as for $z \rightarrow tail = y$ must hold. Notice that variables that are aliased to x, including x itself, are not considered. This is because either the variable is not pointed at by any *tail* and can be ignored, or it is and is therefore dealt with by $z- > tail$. The special cases that hold for $x \rightarrow tail = y$ also hold in this.

5.8 Forward Analysis Sample

To explain how the forward analysis would work, it is useful to give an example. Full details are not given of the analysis, merely an overview. The linked-list `insert` procedure is used, as described in .

```
insert(int i, ptr s) {
    while (s::cons && s->head < i) s = s->tail;
    if (s::nil || s->head > i) *s = *cons(i,copy(s));
}
```

The abstract state before the analysis would be empty. This is then executed over the various statements to find the possible state at each point. Within the `while` guard, the initial piece of knowledge that is determined by `s::cons` is $\varphi(s) = \{cons\}$. This then allows the `s->head` and `s->tail` statements on that line to be proved.

The variable with forward motion in the loop is `s`, and this can be seen from `s = s->tail`. There are no other statements in the loop, so nothing interferes with either `s` or the `tail` path.

After the `while` has executed, then the state must be that at least one of `s::cons` and `s->head < i` is false. How this information is stored is specific to the implementation, although the obvious implementation would be to treat this as no information. Fortunately this is sufficient to check the `if` statement successfully.

The `if` statement first tests if `s::nil`, if this is true execution proceeds to the body of the `if` statement. The second condition of `s->head > i` is only executed is `s::nil` fails, and $\neg$`s::nil` would make $\varphi(s) = \{cons\}$. When `s->head` is selected, this is now safe because of the negative information obtained by the first condition.

The statement inside the `if` has no preconditions related to field selection, and after the state would include $\varphi(s) = \{cons\}$. If the `if` is not taken then the state would include $\varphi(s) = \{cons\}$ and $s \rightarrow head \leq i$. When these two states were joined by the $\odot$ operator, the resulting state would contain $\varphi(s) = \{cons\}$.

From this forward analysis, it can be shown that this procedure is total. In addition the variable s afterwards is of type *cons*.

5.9 Summary

This chapter has presented a design for a forward analyser, including details of the atomic assignment statements for both pointers and integers. Aliasing has been investigated, and two possible schemes have been proposed. Finally, the case of loops has been considered, both fixed pointing of the state, and design of their termination arguments.

Chapter 6:

Backward Analysis

This chapter presents a design for a backward analysis program, including which predicates would be required and how they could be manipulated. Much of the discussion regarding forward analysis also applies to backward analysis, and this is not repeated.

In order to define some of the concepts in a more mathematical way, I have used $[\![x]\!]^{-1}(Q) = Q'$ to mean that if Q is the postcondition to x, then Q' is its precondition. It is also useful to introduce changes in the index, defined as:

$$[\![x]\!]^{-n}(Q) = \begin{cases} Q & \text{where } n = 0 \\ \text{as above} & \text{where } n = 1 \\ [\![x]\!]^{-1}([\![x]\!]^{-(n-1)}(Q)) & \text{where } n > 1 \end{cases}$$

6.1 Terms

The predicate expressions will be written using the normal logic connectives, $\vee$, $\wedge$, $\neg$ etc. along with additional specific terms. These terms need to model the preconditions of safety for each statement, along with any additional notions to derive such safety. The terms used in the logic engine are now introduced, along with their purpose. Each of these terms is only stored while its value cannot be determined, so it is at the $\frac{1}{2}$ logic state. When additional information renders the term either true or false, then it can be replaced with the appropriate literal.

In all the following terms, the variables mentioned may be any program variable, along with any number of selectors. Most of these terms have direct mappings to the information stored in the forward analysis.

6.1.1 Test

The predicate term $test(a, b)$ is equivalent to the Pasta condition a::b. The use of this predicate is primarily for the proof of selectors.

6.1.2 Field

The predicate term $field(a, b)$ says that the variable a has a field b. That is, $a \rightarrow b$ is safe. Clearly the $field$ predicate can be replaced with multiple $test$ predicates, and this is in fact done. However, for the analysis stage, it is useful to think of $field$ as a separate construct.

6.1.3 Alias

The predicate term $alias(a, b)$ means that a and b are aliases of each other. This is useful and replaces the alias analysis involved in the forward analysis method. The advantages are that only the alias information required to be proven is stored, without redundant information.

The trivial case of $alias$ is where $a = b$, i.e. $alias(x, x)$. In this case the alias relation is clearly true, and can be replaced with the literal truth value.

6.1.4 Relation

The predicate $relation(a, op, b)$ is used to say that a numeric relation holds between a and b, where $op \in \{=, <, >, \leq, \geq, \neq\}$. Here a and b can be either variables or numeric literals.

A *relation* term can be evaluated easily when both a and b are numbers by just applying the given operator to them. If a and b are the same variable, then the relation can also be evaluated easily.

6.2 Conditional Statements

The `if` statement can be treated in much the same way as for the forward analysis. The main difference is that in the statement `if (`a`) `b`; else `c`;` it is impossible to deduce the value of a, and therefore determine which branch is taken. Fortunately by using $\Rightarrow$, if the value of a is later determined then the appropriate precondition will remain.

$$[\![\texttt{if (}a\texttt{) }b\texttt{; else }c\texttt{;}]\!]^{-1}(Q) = (a \Rightarrow [\![b]\!]^{-1}(Q)) \wedge (\neg a \Rightarrow [\![c]\!]^{-1}(Q))$$

In this case, if the value of a is completely undetermined even after the entire computation has finished, then it is possible to replace this precondition with a weaker condition, equivalent to that used by the forward analysis when nothing about `a` is known.

$$([\![b]\!]^{-1}(Q) \wedge [\![c]\!]^{-1}(Q)) \Rightarrow ((a \Rightarrow [\![b]\!]^{-1}(Q)) \wedge (\neg a \Rightarrow [\![c]\!]^{-1}(Q)))$$

6.3 Loops

In order to determine that a loop is safe, an acyclic path must exist, and the body of the loop must be safe. In addition the analysis must terminate.

6.3.1 Safety

To analyse a loop it is necessary to take the postconditions, and generate appropriate preconditions by applying the loop statement. This section is concerned with the safety of the statements after the loop, and within the loop body, and not with a proof of termination.

For the statement:

$$[\![\texttt{while (}a\texttt{) }b\texttt{;}]\!]^{-1}(Q) = Q'$$

The first thing to do is to determine the value of Q', given a known number of executions of the loop body b. For example, if this number is 0, then $Q' = Q$, assuming a does not have any preconditions (this assumption is valid in Reduced Pasta). For 1 iteration $Q' = [\![b]\!]^{-1}(Q)$, and after two this would be $Q' = [\![b]\!]^{-2}(Q)$. This means that if n iterations are performed, then:

$$Q' = [\![b]\!]^{-n}(Q)$$

The next step is to determine how many iterations are performed. If 0 iterations are performed, then it must be the case that $\neg a$ is true. For 1 iteration, it must be the case that the first initially a is true, but after an application of b it becomes false, so $a \wedge [\![b]\!]^{-1}(\neg a)$. In general for n iterations, the first n times $\neg a$ must be the case, and the next iteration a is true. The condition for n iterations can therefore be stated as:

$$[\![b]\!]^{-n}(\neg a) \wedge \bigwedge_{i=0}^{n-1} [\![b]\!]^{-i}(a)$$

For the program to be safe, if n iterations are made, then n iterations must be safe. This can be expressed by combining the previous above two equations as:

$$Q' = \forall n \in \mathbb{Z} \bullet n \geq 0 \wedge [\![b]\!]^{-n}(\neg a) \wedge \bigwedge_{i=0}^{n-1} [\![b]\!]^{-i}(a) \Rightarrow [\![b]\!]^{-n}(Q)$$

This precondition only asserts that the operations performed in b and after the end of the loop are safe, not that the loop terminates. In order to generate a finite precondition Q', one approach is to generate a fixed point on this statement. If n is restricted to the set $0..k$ instead of $\mathbb{Z}$ and $Q'^{k=j} = Q'^{k=j+1}$ then by making $k = j$ a fixed point is established which is equivalent to the full precondition.

What happens if this condition does not reach a fixed point? Any precondition P where $P \Rightarrow Q'$ can be used as a conservative approximation of Q'. Using the precondition $False$ would work, although admittedly at the cost of making proving safety unlikely.

One (possibly convoluted) example of where fixed-pointing would not occur is deleting all elements from a linked list which occur before the maximum element. A safe implementation, ignoring termination issues, is shown in Figure 6.11. For this piece of code, the second loop would generate an infinite precondition that would not reach a fixed point. However, this precondition is correct, using the loop above it. It would be possible to rewrite this function into one involving only one `while` statement, and this new piece of code would be provable.

Figure 6.11: Safe deletion of all elements before the maximum

```
ptr x = cons(0, ...); int Max = -1;

ptr i = x; while (i::cons) {
    if (Max > i->head)
        Max = i->head;
    i = i->tail;
}

while (x->head > Max)
    x = x->tail;
```

Another example of a piece of safe code that generates a non-fixed-pointing `while` precondition is shown in Figure 6.12. In this sample, the condition for loop safety is infinite, however the loop is never executed. The next question is what to do about these loops. Both these examples can be rewritten into a safe format, in the second case by removing the `while` statement entirely. Because of the possibility of rewriting, fixed pointing is probably a reasonable method to choose.

Having decided to use fixed pointing of the precondition, the next question is at what number to bound the search for the fixed point. The smaller this number the quicker the program will detect loops that do not fixed point, but this may result in certain safe loops being erroneously reported as having an infinite precondition. This number should therefore be determined by experimentation.

Figure 6.12: A non-fixed-pointing precondition

```
ptr a = nil(); ptr b = ...;

while (a::cons)
    b = b->tail;
```

6.3.2 Forward

The next step after proving safety of selectors is to prove the loop terminates, by finding an acyclic path. To find an acyclic path, it is useful to introduce an ordering over heap cells. This ordering, $a \succ b$ means that b can be reached from a by following only acyclic selectors. In addition, due to the properties of acyclicity, this means a cannot be reached by b, following acyclic selectors.

$$a \succ b \Rightarrow b \not\succ a$$

For implementation this definition would have to be complicated in order to account for the possibility of multiple `acyclic` statements within one signature. While this would add to the implementation complexity, it is easy to expand from one $\succ$ operator to many. For the discussion of the theory, one acyclic set alone is assumed.

6.3.3 Termination

To prove termination is valid, there must be one pointer such that on all iterations, the value of the pointer is a successor using $\succ$, of its original value afterwards. A pointer could be any variable, followed by any number of selectors. If such a variable exists, then there will be one that is mentioned within the loop. If the variable that is updated in this manner is done so via an alias, then the alias would be the appropriate variable. This means that there are only a finite number of candidate pointers.

A way to check that a pointer points to a forward point after the loop is to propagate the variable over the statements within the loop. If regardless of the branches taken, the variable points at a forward location afterwards, this variable proves termination for this loop. Care needs to be taken that any assignments do not destroy this property as they are executed. In particular starred assignment may destroy this property.

6.4 Statements

This section details exact equations that specify how postconditions are transformed by the various atomic Pasta statements.

6.4.1 Simple variable assignments

One common effect of assignment is to rename the variables. For example, in $[\![x = y]\!]^{-1}(Q) = Q'$, it is possible to replace all terms in Q which mention x with y. I have denoted this as $Q^{x \mapsto y}$. In particular, the terms for which this is relevant is the *test*, $field$, *alias* and *relation* terms. Using this notation, the effect of the above statement can be given as:

$$[\![x = y]\!]^{-1}(Q) = Q^{x \mapsto y}$$

6.4.2 Assignments from a field

The effect of the statement $x = y \rightarrow field$ is similar to $x = y$, but an additional safety precondition is generated to check the selector access is safe.

$$[\![x = y \rightarrow field]\!]^{-1}(Q) = Q^{x \mapsto y \rightarrow field} \wedge field(y, field)$$

6.4.3 Assignment to fields

$$[\![x \rightarrow field = y]\!]^{-1}(Q) = Q^{\forall z \bullet alias(x,z) \Rightarrow z \rightarrow field \mapsto y} \wedge field(x, field) \wedge x \succ y$$

Note the transformation of Q. The $\forall$ seems to suggest an infinite number of terms may be generated, but this is not the case. The only terms that need to be included as values for z are the pointers that are included in the list of terms. As a more concrete example, for the single term $Q = test(a \rightarrow field, cons)$, the expanded version would be:

$$\begin{aligned}[\![x \rightarrow field = y]\!]^{-1}(test(a \rightarrow field, cons)) = \quad & (alias(x, a \rightarrow field) \Rightarrow test(y, cons)) \wedge \\ & (\neg alias(x, a \rightarrow field) \Rightarrow test(a \rightarrow field, cons))\end{aligned}$$

Note also the inclusion of $x \succ y$, which is required to maintain the acyclicity. In particular, note that x and not $x \rightarrow field$ is compared to y. This proves that $x \rightarrow field = x \rightarrow field$ maintains the acyclicity property, as indeed it does.

6.4.4 Starred Assignment

$$[\![*x = *y]\!]^{-1}(Q) = Q^{\forall z \bullet alias(x,z) \Rightarrow z \mapsto y} \wedge (\forall z \bullet alias(x, z \rightarrow field) \Rightarrow z \succ y)$$

The starred assignment is very similar to the assignment to a field. The only addition is that now for all variables aliased to x, the acyclic constraint must be preserved. Unfortunately this would expand to an infinite number of preconditions, requiring all possible z aliases to be checked, even for values of z not mentioned in the procedure.

One way to permit the starred assignment would be to introduce the universal quantifier to the predicate logic system. This would complicate all the other expressions, requiring them to deal with quantifiers, and make implementation harder.

Another method is to remove the quantifier, replacing the condition with an alternative one, that implies the original condition. One replacement condition is:

$$[\![*x = *y]\!]^{-1}(Q) = Q^{\forall z \bullet alias(x,z) \Rightarrow z \mapsto y} \wedge (\neg alias(x,y) \Rightarrow x \succ y)$$

In this expression, $x \succ y$ is used, as this implies $\forall z \bullet alias(x, z \rightarrow field) \Rightarrow z \succ y$. There are various cases in which the new condition will fail, while the old one succeeds, but the most obvious is when $alias(x, y)$ is true. In this situation, the $*x = *y$ statement has no effect, and therefore $alias(x, y)$ can be used to guard the $\succ$ condition.

The new expression has a finite number of terms in it, at the cost of slightly reduced power compared to the original. While some statements accepted by the original expression are now rejected, this does not appear to include any statements actually used in programs.

6.4.5 Allocating Assignments

A statement $x = new()$ has no preconditions, and allows any $alias(x, ?)$ statements to be discharged, since this new node is clearly not aliased to anything. It also allows $test(x, ?)$ statements to be discharged.

6.5 Predicate Logic

This section concerns how to store the predicates at each stage. The most simple format is in the form in which they are generated. The main problem with this is that the information available at any stage of the computation to the user will be in an incredibly bloated format. The other problem is that loops require fixed pointing of the precondition, and by not having a standard representation, fixed pointing is not as easy.

When choosing a format for propositional logic, the two standard forms are disjunctive normal form, and conjunctive formal form. This is achieved by removing statements such as $\Rightarrow$ and replacing $a \Rightarrow b$ with $\neg a \vee b$. Brackets are multiplied out as required, and this generally leads to repeated subexpressions.

An important concern here is the $field$ predicate, which can be replaced by the disjunction of all the types which contain such a field. However, doing this does not allow negative information to accumulate. The fact that a variable is *not* of a particular type does not solve any terms. The reverse, saying that the type is not any of the types not containing the field allows both negative information to accumulate (by solving a term), and positive information (by solving all terms). So the ideal expansion, where $Type^{\neg field}$ is the set of all subtypes not containing a particular field, is:

$$field(x, selector) = \forall y \in Type^{\neg field} \bullet \neg test(x, y)$$

Having established this as the format for checking selectors, which is very common precondition, we need to design a predicate representation that can encode this condition efficiently. Using conjunctive normal form for the expression would result in a very large number of terms, when an outer operator such as $\wedge$ was expanded. For this reason, disjunctive normal form is an appropriate choice.

Using disjunctive normal form, what optimisations can we apply and at what stage? Various boolean relations, in particular idempotence, can be applied. Deciding what level of optimisation to apply and at what stage is therefore deferred to the results section, where measurements can be made.

6.6 Summary

In this chapter I have discussed the design decisions specific to backward analysis, covering the terms required and the predicate expression. Details have also been given of the transformations individual statements make to the predicates.

Chapter 7:

Implementation and Testing

This chapter covers the details of implementation, how the static analysis program is constructed, and the testing carried out. Selected highlights of the program are included in Appendix .

7.1 Choice of Analysis Method

I have implemented both forward and backward analysis, as described in Chapters 5 and 6. Following experimentation with both types, I decided that backward analysis was more appropriate for this particular task. Details of the performance of the forward analysis can be seen in section , along with reasons for prefering backward analysis. I have included some design details related to forward analysis, however the majority of this chapter is concerned with backwards analysis.

7.2 Language

I chose to implement my analysis program in Haskell [20], primarily because the existing parser and implementation for the Pasta programming language are written in this language. In addition, the mathematical model of the static analysis can be mapped easily into Haskell.

The primary disadvantage of using Haskell is that the destructive pointer assignment in Pasta would benefit greatly from a language which exposed the power of references. As a result alternative mechanisms are required to achieve the same effect. This is mainly of note in the forward analysis.

7.3 Abstract Representation

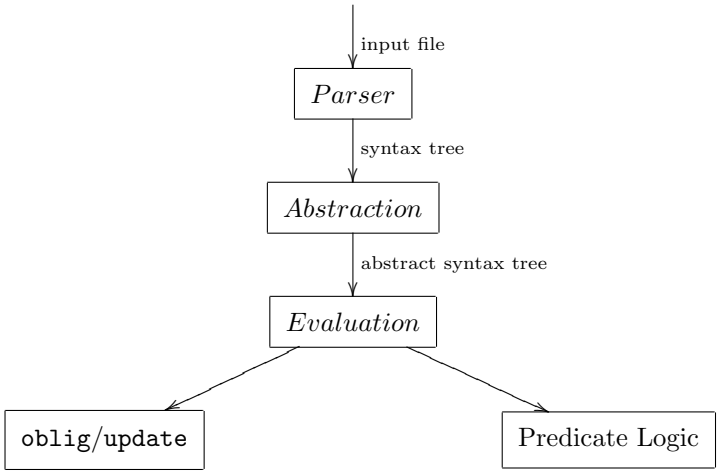
As discussed in Chapter 4, the Pasta language contains many constructs that can either be removed or simplified, and the design chapters considered Reduced Pasta. For the implementation, breaking Pasta down to the Reduced level given is not necessarily practical, as many of the abstractions complicate the structure of the program.

One particular example of the abstraction tradeoff is the `while` statement. The simplified `while` removes the condition out of the guard, which decreases complexity for mathematical analysis, but increases it for implementation. However, if the `while` statement is retained with its guard, the short-circuit boolean operators `&& ||` must be implemented directly. Having implemented them for the `while` statement, the code and its inherent complexity can be reused for the `if` statements.

A disadvantage of any abstraction process is that the knowledge determined by the static analyser is harder to reintegrate with the standard abstract syntax tree for the full Pasta program. This means that certain applications, for example optimising compilers as discussed later, would require more work. However, I feel that the benefit of having simpler code outweighs this, certainly for an initial implementation.

One particular note in relation to the abstractions is that temporary variables have been minimised, in order to avoid a global counter. All variables are given a number related to their scoping, so variables in the outermost scope are named `varname1`, within one statement `varname2` etc. This is also used for procedures that are expanded inline, where the code is considered to be with a block. Where additional temporaries are needed, for example parallel assignment, a leading underscore is used in addition to the standard variable. This is valid, because in standard Pasta programs a variable name may not contain either an underscore or a digit, and hence all the generated names are unique.

Figure 7.13: Program Structure



7.4 Program Design

The analysis program is designed to be modular. The analysis is detached from the related tasks such as the abstraction and the predicate solver. In order to implement the solver, two particular functions are used to generate the preconditions, namely `oblig` and `update`. The `oblig` function takes an atomic statement, and returns the safety preconditions. The `update` function takes both a statement, and a single predicate term, and generates a modified predicate term or terms. Special case solvers are used for both the `if` and `while` statements.

An object diagram of the program is shown in Figure 7.13. In this diagram the program to analyse flows down from the top to the bottom. The Evaluation stage uses both the `update` and `oblig` procedures, in addition to the predicate solver.

7.4.1 Short Circuit Boolean Operators

The short-circuit boolean operators, `&&` and `||`, are implemented in a separate function. The statement `a || b` in particular is defined as being the case when either `a` is true, or `a` is false, and `b` is true. This raises the need to determine the value of `not a`, remembering that the Pasta language has no such `not` keyword or `!` operator. To create a `not` of a numerical relation is easy enough, and simply requires inverting the operator. A test `x::cons` can be replaced with $n - 1$ terms, where n is the number of subtypes, being all the subtypes but the original one. However, in order to reduce the number of terms, I have added the `not` symbol into the language at the abstraction level. This construct is particularly easy to process, as it maps trivially to the predicate logic $\neg$ term.

7.4.2 Obligations as Predicate Trees

The `oblig` function takes a statement, and generates the obligations for it as a predicate tree. If the predicate tree generated is true, then the statement is safe. For an example, if `x = y->tail` is the statement, then $\neg test(y, nil)$ is the generated predicate.

As a particular example, the statement $x = y$ is dealt with by the code in listing 7.1. The obligations are

Listing 7.1: Implementation of `oblig`

```
oblig prog (Assign l1 l2) = PAnd [obLoc prog l1, obLoc prog l2]

obLoc prog loc = PAnd (map f (inits loc))
    where
        f [] = Pass ""
        f [_] = Pass ""
        f loc = obSelector prog (init loc) (last loc)
```

Listing 7.2: Implementation of `update`

```
update prog (New l n) p = case p of
    s@(Test l' n') -> if l == l' then porf (n == n') (show s) else Proof p
    s@(Alias x y) | x == y -> Pass (show s)
    s@(Alias x y) | x == l || y == l -> Fail (show s)
    x -> Proof x
```

that the two variables are both valid down all their selectors. These obligations are generated by taking all the selectors and applying the `obSelector` function.

7.4.3 Predicate Updating

The `update` function maps a particular term over a statement. An example of this is the term $test(x, nil)$, when mapped over `x = y`, is changed to $test(y, nil)$. However, while the input term is just one single atomic predicate, the output term does not have to be. For example, $test(q \rightarrow tail, nil)$ when mapped over `x->tail = y` generates $(alias(x, q) \Rightarrow test(y, nil)) \wedge (\neg alias(x, q) \Rightarrow test(q \rightarrow tail))$.

One equation in the definition of `update` is given in Listing 7.2. This equation handles the $x = new()$ statement, with `n` being the subtype and `l` being the variable. This particular function discharges all *test* terms related to the variable x by directly checking the type, and also deals with *alias* relations where either variable is x.

7.5 Loop Design

The question of how to implement looping is of particular difficulty. The basic problem is to find a pointer which points to a successor after the loop has executed. The method that is used is to generate all possible variables that might satisfy this, along with all possible sets of acyclic variables, and take the cross product. The disjunction of all of these statements is then taken, with each condition being of the form $forward(variable, variable, selectors)$. In the actual implementation `FwdLoop` also has an integer variable to denote the depth – this is only used for dealing with nested loop statements. The `update` function mentioned above then performs variable renaming on only one of the variables.

After a loop has been executed once, all $forward$ terms are examined. Any that have shown forward motion are retained for the next iteration, and any which do not are falsified. This is repeated until the a fixed point is reached, at which time all remaining $forward$ terms are passed. This results in the correct behaviour for finding forward motion. If no forward terms remain, then the whole condition fails. Even if the $forward$

terms have become distributed throughout the entire expression, for example via $alias(\ldots) \Rightarrow forward(\ldots)$, this method still works.

This implementation is particularly nice, because the `update` function can be reused to perform most of the work. The main disadvantage however, is the number of pointers that are checked for forward motion, which can become quite big. Most of these pointers are rejected in the first iteration, so this does not prove too much of a problem.

The vast number of $forward$ predicates complicates the output traces considerably in some cases, and so the program was altered to produce more effective output. First the loop is analysed once using all the possible variables, and then all those that do not show any forward motion are discarded, and the loop is reanalysed from the beginning using only the remaining variables. This does not affect the result of the analysis. It takes additional time, but the result is more usable for a human reader.

7.6 Testing

As always, testing the program to check it performs correctly is of paramount important. There are various testing methods available, and different approaches are appropriate for different sections of the program.

7.6.1 Random Testing using QuickCheck

The program QuickCheck [10] is written in Haskell, and is a tool for testing Haskell programs automatically. QuickCheck generates large numbers of sample expressions, and checks that properties supplied by the tester hold under all these cases. QuickCheck can be used as a reasonable program tester in functional programs because the lack of global state allows the testing of each function to be performed in isolation without worrying about the state of the system. To apply QuickCheck in a program it is necessary to give a mechanism for QuickCheck to generate sample expressions, and to give various properties for it to check against.

I considered using QuickCheck to test the entire analysis. The main problem is generating random programs. It is entirely possible to generate a random program, and by imposing certain restrictions on it valid Pasta programs, with appropriate typing, can be generated. The problem however is that the vast majority of programs will not be safe and will not terminate. Determining which category a random program falls into requires manual intervention, so automated testing cannot be performed. Despite this, checking random programs would help to suggest the analysis itself did not crash and always terminated, but this would require a lot of effort to implement for relatively little gain.

One particularly complex area of the program is the predicate solver. This takes a predicate expression, puts it into disjunctive normal form, and simplifies it. The important properties about this transformation are that the original expression is equivalent to the final one, and that the final one is in disjunctive normal form. Both these are perfect candidates for checking with QuickCheck, and this was done.

Random Predicates The first task when using QuickCheck is to randomly generate various expressions that can be used as test cases. The proof engine in the analysis program allows the use of the symbols $\Rightarrow \wedge \vee \neg$ along with the literals $true$ and $false$, and any individual terms, for example $alias$ and $test$. The data structure is parameterised by the terms given, and for the solver, these are treated as the $\frac{1}{2}$ logic value.

To generate random expressions, I defined the atomic terms to be either `Proof True` or `Proof False`. These both have no meaning to the solver, but can be evaluated to reach concrete boolean values at the end. These were generated along with the allowed logic symbols.

Listing 7.3: QuickCheck Disjunctive Normal Form property

```
prop_dnf :: ProofBool -> Bool
prop_dnf (PB x) = f 0 (simplify x)
    where
        f i (POr x) | i < 1 = all (f 1) x
        f i (PAnd x) | i < 2 = all (f 2) x
        f i (Not x) | i < 3 = f 3 x

        f _ (Pass _) = True
        f _ (Fail _) = True
        f _ (Proof _) = True
        f _ _ = False
```

Listing 7.4: QuickCheck correctness property

```
prop_eq :: ProofBool -> Bool
prop_eq (PB x) = f x == f (simplify x)
    where
        f (POr x) = any f x
        f (PAnd x) = all f x
        f (Not x) = not (f x)
        f (Imp x y) = not (f x) || (f y)
        f (Pass _) = True
        f (Fail _) = False
        f (Proof x) = x
```

The output after simplification should be in disjunctive normal form, and this is checked by using the Quick Check property given in listing 7.3. This code maintains a variable, `i`, which can only ever increase. As the various connective symbols are reached, this counter is increased so that lower down connectives cannot be duplicated.

The correctness of the simplification is checked by evaluating the pending values into real values, and then applying all the operators, as shown in Listing 7.4. Just because `prop_eq` passes, this does not mean that in this particular case the `simplify` function is correct. Because only two pending values are used, and only two outcomes can result, it is quite possible that the `simplify` was wrong, yet correct by chance. However, if this property fails then it is clear that the `simplify` function is flawed. As a result, a larger number of these tests need to be performed, however because this is an automated process, this is not of particular concern.

7.6.2 Example Testing

The most obvious method of testing the program was on the sample Pasta programs available to me. A number of samples exist, including an ordered linked list, a queue and various tree structures. For each of these samples, the analyser was run over it, and the results compared with manual analysis. Where failures were found, these were investigated.

More details of the examples, their results, and a full analysis can be found in the following chapter.

7.6.3 Regression Testing

The final method of testing used was regression testing [3]. As failing examples were found, the minimal failing example was extracted and placed in a separate file, shown in Appendix . Also, as the development progressed and new features were able to be validated, small pieces of code exercising these features were created and placed in the same file. By the end of development, a corpus of over 20 snippets of code was available. The entire test suite could then be run automatically, and compared against the expected results.

7.7 Summary

The analysis program was implemented in accordance with the design section, using Haskell as an implementation language. Due to the mathematical nature of both the design and Haskell, the majority of the implementation is just basic translation. Testing was done on the analysis program, which allow a level of confidence to be asserted in its correctness.

Chapter 8:

Results and Evaluation

The program was implemented, and then executed on the various sample programs. This chapter discusses the results, and any issues that were raised by the resulting implementation.

8.1 Criteria for Evaluation

The first task when is to define some criteria by which to assess the program. The aim of the static analysis tool was to prove that a Pasta function is total. The tool is designed to generate proofs, so for any function that cannot be proven, the tool will report a failure. This means that the primary evaluation criterion is that the tool is indeed correct, and that no unsafe programs are reported as safe.

The program solves the halting problem for a restricted set of cases, and this means there are an infinite number of safe programs for which the program will report failure. A much more interesting question is how many programs that people will actually want to write, and can be expressed naturally in Pasta, are erroneously reported as failures.

Given that there exist safe programs that cannot be proved safe by the analyser, in order to use the tool effectively on a larger project, these procedures would need to be rewritten in such a way that proof is possible. Therefore another criterion is *how much additional restructuring or annotation is needed* to take a safe but failing example, and obtain a proof of correctness.

If the analysis fails, then the next question will always by "why?". This means that another important criterion is the *quality of any error report*, a terse `No.` is insufficient, as this does not allow the programmer to take corrective action. The ideal analysis would pinpoint the exact statement which failed, along with detailed reasons for its failure.

8.2 Linked List

The most basic pointer based data structure is a linked list, and therefore makes a good candidate to analyse first. The linked list sample given in Section was analysed. The output from the backward analysis tool is shown in Listing 8.5. [1]

The system uses square brackets to show the preconditions at each stage, as they are hoisted over the statements. The only possible source of confusion could be from the `if` statement – just before both the `if` and `else` clauses the combined precondition is listed.

The first thing to note, is that the condition at the bottom of the sample is $True$, this is the initial condition for the system – there are no preconditions to discharge after the function. The next thing is the $True$ at the top, this means that the program is safe. By looking at just the program, and not the conditions at all, the abstract representation of Pasta in the program can be seen.

The `while` statement analysed at the top shows that the loop terminates, because it traverses down the acyclic path composed of `tail`, using the variable `s`. The other point to note is that the loop is safe on its own, i.e. the precondition is $True$. This is because any safety preconditions, namely that $s :: cons$, are

[1] Editorial changes have been made to the output of the program. These are purely formatting changes – no formulae have been altered and the original meaning and layout is preserved.

Listing 8.5: Output of the analysis program for linked list insertion

$[True]$
`Forward down: s::tail`
`Fixpoints on:` $[True]$
while `((s::cons) && (s->head Less i)) {`
    $[forard(s, s, tail) \vee forward(s \rightarrow tail, s \rightarrow tail, tail)]$
    $[(\neg s{::}nil \wedge forward(s, s, tail)) \vee (\neg s{::}nil \wedge forward(s \rightarrow tail, s \rightarrow tail, tail)]$
    $[(\neg s{::}nil \wedge forward(s \rightarrow tail, s, tail)) \vee (\neg s{::}nil \wedge forward(s \rightarrow tail \rightarrow tail, s \rightarrow tail, tail))]$
    `s = s->tail`
    $[forward(s, s, tail) \vee forward(s \rightarrow tail, s \rightarrow tail, tail)]$
`}`
$[True]$
if `((s::nil) || (s->head Great i)) {`
    $[True]$
    $[True]$
    `_s = new(cons)`
    $[\neg_s{::}nil]$
    `_s->head = i`
    $[\neg_s{::}nil]$
    if `(s::nil) {`
        $[\neg_s{::}nil \vee \neg s{::}nil]$
        $[\neg_s{::}nil]$
        `_s->tail = new(nil)`
        $[True]$
    `}` **else** `{`
        $[(\neg_s{::}nil \vee s{::}nil)]$
        $[(\neg_s{::}nil \wedge \neg s{::}nil)]$
        `_s->tail = new(cons)`
        $[(\neg_s{::}nil \wedge \neg_s \rightarrow tail{::}nil \wedge \neg s{::}nil)]$
        `_s->tail->head = s->head`
        $[(\neg_s{::}nil \wedge \neg_s \rightarrow tail{::}nil \wedge \neg s{::}nil)]$
        `_s->tail->tail = s->tail`
        $[True]$
    `}`
    $[True]$
    `s *=* _s`
    $[True]$
`}` **else** `{`
    $[True]$
    $[True]$
`}`
$[True]$

discharged in the loop guard. This is a very common pattern for loop construction, and the vast majority do indeed fixed point on $True$.

8.3 Queue Analysis

After analysing a linked list, a good choice of data structure is the queue. This incorporates a linked list of elements, but also allows insertion at the end, using starred assignment. The signatures of the methods are given in Figure 8.14. This program defines a queue, where `enqueue` adds an element `i` to the end of the queue, and `dequeue` removes the first element in the queue. The full code for these methods can be found in Appendix .

The first thing to note about this example is the `makequeue` function. This performs initialisation required for the queue data structure. The other thing is the `queue` subtype, this is essentially a composite object, storing a pointer to both the head and the tail. It is the programmers intention that the `q` parameter to both `enqueue` and `dequeue` is of type `queue`, and was constructed with `makequeue`. This is not however explicitly stated in the code, merely as part of the documentation of this interface.

Figure 8.14: Signatures of the methods for a queue

```
queue acyclic(tail) {
    nil();
    cons(int head, ptr tail);
    queue(ptr front, ptr rear);
}

makequeue(ptr q);
enqueue(int i, ptr q);
dequeue(ptr q);
main();
```

I ran the analysis program over the four different procedures. The results were as follows.

8.3.1 Make Queue

The `makequeue` procedure passes. This is not surprising, as the parameter `q` is not actually used, other than to assign to. In a language such as Ada, `q` would be an out parameter. Since the function does not depend on `q`, and everything else is known, success is expected.

8.3.2 Main

The `main` procedure also passes, this is an example program which inserts and deletes items from the queue. All the interfaces are called following the appropriate conventions, for example `makequeue` is called first. As the program conforms to the interfaces, it is to be expected that the program is safe, as is proven.

8.3.3 Dequeue

The `dequeue` procedure deduces the precondition $[\neg q{::}cons \wedge \neg q{::}nil]$ which can be rewritten as $q{::}queue$. This is reasonable, as the `dequeue` statement is only applicable to `queue` subtypes. In order to change the program to make this procedure valid without any precondition, the body could be guarded with an `if (q::queue) { ... }` statement. When this was done, the result of the analysis was a precondition of $True$.

8.3.4 Enqueue

The `enqueue` procedure deduces the most complex precondition of all the procedures, being:

$$[\neg q{::}cons \wedge \neg q{::}nil \wedge \neg alias(q, q \rightarrow rear)]$$

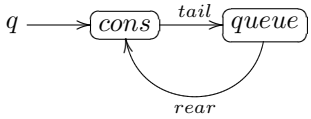
The first two terms are equivalent to q::*queue* as for `dequeue`, and could be removed by introducing an `if` statement to guard the procedure body. The *alias* predicate however is not so obvious. To see the reason for this predicate, it is necessary to delve into the code behind the procedure.

```
enqueue(int i, ptr q) {
    *q->rear = *cons(i,copy(q->rear));
    q->rear = q->rear->tail;
}
```

This code creates a new node, using `cons`, which stores the value to be inserted. It then overwrites the existing `rear` node, with the in-place starred assignment, thus ensuring that all the `tail` pointers are still valid. The `tail` for this new node is a copy of the old tail node, which will in fact always be `nil` for valid structures and hence `copy(q->rear)` could have been replaced with `nil()`. The next statement then updates the rear to point to this newly created `nil` node.

Now let us consider what happens if the precondition suggested by the analysis fails to hold. That is, $alias(q, q \rightarrow rear)$. Before the computation starts:

After the execution of the statement `*q->rear = *cons(i,copy(q->rear));`:



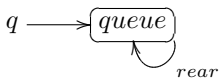
Now the field selection `q->rear` is no longer safe, as `q` is of type `cons` and thus does not have a `rear`. This would cause a crash.

It is possible to rewrite this procedure, in such a way that the results are identical when the data structure is valid. If the structure is corrupt, then the procedure will not perform as intended, but will not cause a program crash. The new version is:

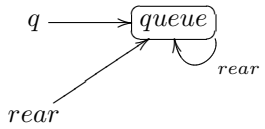
```
enqueue(int i, ptr q) {
    ptr rear = q->rear;
    q->rear = copy(q->rear);
    *rear = *cons(i,q->rear);
}
```

The result of analysing this program is that the condition about aliasing is removed, and merely the q::*queue* condition remains, as expected. It is interesting to see what happens using the new code when $alias(q, q \rightarrow rear)$.

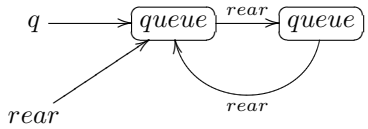
The initial state is the same, namely:



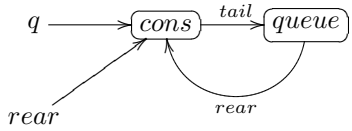
After the `ptr rear = q->rear` statement:



The statement `q->rear = copy(q->rear)` results in:



And the final `*rear = *cons(i,q->rear)` gives:



While it is clear that this data structure is not a valid queue, the procedure does perform correctly on valid queues, and safely even on invalid queues. This satisfies the requirement of totality, but not that the procedure performs according to specification. Hopefully the graph analyser would detect a situation such as this, being outside the scope of this project.

8.4 Tree

Another data structure based on pointers is a binary tree, in which each number is held in order: all children on the left of a node are lower, all to the right are higher. The type signature and function signatures of this sample are shown in Figure 8.15.

In this example, all the procedures end up with the precondition $True$. The analysis does not present any problems, and is dealt with easily by the program. The only point that may be of note is that analysis of all the loops reaches a fixed point with the precondition $True$.

8.5 Threaded Tree

Another pointer data structure is the threaded tree, this data structure differs from the previous ones by maintaining two sets of pointers to each node. This data structure can be thought of as a cross between a doubly linked list and a binary tree, nodes are stored both in order of insertion and in numerical order. The other point to mention is that the implementation of the threaded tree is significantly more complex than the previous samples, and highlights some of the limitations of the analysis. The method signatures are given in Figure 8.16 and the original implementation is available in Appendix .

Figure 8.15: Method signatures for a binary tree

```
tree acyclic(left, right) {
    leaf();
    fork(ptr left, int n, ptr right)
        ;
}

insert(int i, ptr t); delete(int i,
    ptr t); main();
```

Figure 8.16: Method signatures for a threaded tree

```
thread {
    leaf();
    branch(ptr prev, ptr left, int n
        , ptr right, ptr next);
    root(ptr first, ptr top, ptr
        last);
}

maket(ptr t)
insert(int i, ptr t)
delete(int i, ptr t)
main()
```

The threaded tree implementation has a container type, called `root` which acts very much in a similar way to `queue` in the queue sample. The `maket` procedure performs

the required initialisation, very much like `makequeue`. Because of this similarity, many of the remarks about the queue data structure hold. In particular, both `insert` and `delete` assume that `t` is of type `root`.

In the threaded tree, the assumption is also made that there is only one `root`. For example the following code can be extracted from the `insert` procedure.

```
if (t->top::leaf)
    ...;
else {
    t->right ...
}
```

Note that in the `else` branch, the code has assumed `t->top::branch`, because the documentation requires `t` to be the only `root`. If this fragment was to occur inside a loop then an infinite precondition would be generated, which would cause the analysis to fail. Fortunately, in this particular example, an explicit `::branch` test is performed inside loops.

The next complexity within the threaded tree is the use of trichotomies to check for subtypes.

```
while (t->n < i && t->right::branch ||
       t->n > i && t->left::branch)
{
    if (t->n < i) t = t->right;
    else t = t->left;
}
```

This code checks the value of `t->n` against `i`, and also checks that the required field is a branch. The fact that `t->right` is a branch is then not required until the next iteration, when `t->n` is called in the guard. This code relies on the fact that only one of `t->n < i` and `t->n > i` can be true, and hence the following condition must also be true for the loop to have been entered.

The analysis program is capable of checking for this when trichotomies are enabled. If they are disabled, then this code generates an infinite number of preconditions related to the `t->n` field selection, and fails the analysis.

While this example is particularly complex, the analysis program does manage to prove all procedures, albeit after modifications have been made. While the modifications are not extensive, they did require some thought as to whether they would alter the meaning of the code. This highlights the disadvantage of modifications, in proving the safety of the code its correctness may be lost.

8.6 Forward Analysis

The forward analysis program, as described in the design section, was also implemented. The performance of this tool is now discussed, although the backward analysis showed itself to be superior in a large range of cases, so forward analysis is only covered in brief.

The first point to mention about the forward analysis is that the program was significantly more complex to write. This is partially because of Haskell being more suited to backward analysis, but even taking this into account, a larger number of modules were required – each of which was significantly more complex.

For the backward analysis the only portion that required custom implementation was the `oblig` and `update` statements. All other elements of backward analysis implemented standard functionality that could be found in tools that did not do analysis.

The exact figures for the size of each implementation are given below:

Analysis method	Lines of code	Number of modules	Size of source
Backward analysis	930	9	32.9Kb
Forward analysis	1583	16	49.1Kb

The forward analysis was able to prove both the linked list and tree examples correct, and over and above the backward analysis, was able to produce a generic state at each step. While this generic state was useful for understanding the code, and the decisions reached by the analysis, in this setting it had only marginal value. If this information could have been used by other processes, for example by an optimising compiler, then this would have been a big point in favour of forward analysis.

One problem with the forward analysis was particularly highlighted during execution on the queue data structure. In this case, where the backwards analysis gave clear and concise results, the forward analysis simply failed. The reason for this is that the abstract state of the program was not able to discharge the precondition, and, unlike the backward analysis, it could not tell what additional condition safety depended on. In the end, it took guess work and experimentation to determine what sequence of inputs would result in a crash. However, with the amended version of the queue, the program could be proved.

The final drawback in the forward analyser is the aliasing information. As shown in Chapter 4, if sufficient aliasing information cannot be determined, then large portions of the abstract state have to be discarded. The perfect forward analyser would therefore store everything that may be needed in the future, but this would require unbounded amounts of storage. A compromise has to be achieved, which limits the program severely. The backward analysis does not suffer from this problem, as essentially it first generates what aliasing information would be useful, and then tries to see if it is true, removing redundant storage.

For these reasons, backward analysis appears to give superior results for the problem of determining totality.

8.7 Disjoint Subtypes

One particular feature of Pasta that has complicated the analysis are the container subtypes, for example `queue` in the queue example. In the original Pasta there is a single pointer data type, with subtypes for each constructor. If these subtypes were split into more than one data type then an alternative definition for queue could be:

```
queue {
    queue(node front, node rear);
}
node acyclic(tail) {
    nil();
    cons(int head, node tail);
}
```

All `ptr` keywords would then be removed, and replaced with one of these specialised subtypes. The type of each variable could then be determined by the type checker, and this information could be used to remove preconditions. The program would also better represent the intention of the user. In fact, in this particular case the $alias(q, q \rightarrow rear)$ precondition would be discharged, because q is of type `queue`, while $q \rightarrow rear$ is of type `node`, hence they are not aliased.

8.8 Runaway Non-termination

An interesting result occurs when a loop does not have an acyclic path to use for forward motion. Consider the linked-list example, but without the `acyclic(tail)` statement in the signature. The following piece of code cannot be proved to terminate:

```
ptr q = ...;
while (q::cons) {
    q = q->tail;
}
```

This statement does not have an acyclic path, and in fact, if no `acyclic` definitions are given, then it is impossible to prove a `while` statement correct. However, the analyser as originally designed tries to prove that the loop still terminates. The generated precondition is an infinite chain:

$$q{::}nil \vee q \rightarrow tail{::}nil \vee q \rightarrow tail \rightarrow tail{::}nil \vee \ldots$$

This precondition is essentially the analysis program trying to prove that the loop comes to an end, without using an acyclic argument. The problem with this is that clearly the condition does not reach a fixed point. If this condition can be satisfied, then the number of iterations of the loop can be determined, and there is likely to be a logic bug somewhere in the code.

To counteract this infinite precondition, and to make the loop fail gracefully, it was necessary to remove the initial implications from the loop precondition. These were the statements that only required the loop to be safe if the iteration was taken. However, the tail-chasing example shows that a more appropriate course of action may be to assume the loop is always able to be executed.

The analysis with the implications removed is strictly less powerful than the initial one, in terms of what can be validated. The only benefit of the second method is that the failure mode is more clear, and allows better pinpointing of the error.

8.9 Performance

During the design and implementation of the static analyser, performance was never a key goal. However, it is useful to see what the existing performance of the system is. To test the speed of the system, I ran the analysis over the queue data structure discussed above. The analysis took 29 seconds, 17348614 reductions, 39750014 cells and required 16 garbage collections.

Almost half a minute for such a short program is quite poor performance. By executing various procedures in isolation, it is possible to determine that over 27 seconds of this runtime are spent in the predicate logic engine. All of this time is spent simplifying the logic, and converting to disjunctive normal form.

In order to investigate why this was taking such a large amount of time, I disabled the simplification logic, means that no time was spent in the logic engine. With this change, the runtime increased significantly as did the memory requirements. The reason for this is that many terms are generated, most of which are removed by the simplifier. Without this simplifier, the precondition increases in size dramatically, and as each statement is mapped over every predicate term, this increases the execution time.

During the development process, it was necessary to optimise the simplification engine for speed on more than one occasion. As the execution time increased to a degree where simple examples became infeasible, the simplifier was improved and this brought the execution times down dramatically.

The performance of the tool is currently unsatisfactory. However, since this is not an area that has been investigated in any depth, this is not unreasonable.

8.10 Overall Results

The analysis program produced does indeed correctly determine whether a Pasta program is total or not. Where examples are shown that do not pass the analysis, this is usually because of a legitimate safety concern. Some programs require modification to allow proof (most commonly via the use of the `acyclic` keyword), although this is usually relatively simple.

Where errors are raised they tend to be of a targeted nature, with a failing precondition that can be used to engineer appropriate test cases. The output at each stage of the process can be viewed, although disjunctive normal form is not necessarily the most appropriate for a human reader.

While there are many safe programs for which termination cannot be proved, these do not appear to correspond to any common programming style or structure.

If any one area of the program was to be described as weaker than the others, it would be the loop analysis. While in practice a fixed point seems to occur at $True$ in almost every common case, this is not something that can be relied upon. Also the problem of fixed pointing brings up the question of how many iterations should be performed before failure is assumed. In this program, I found by experimentation that 3 iterations was sufficient. Of course, it is always possible to engineer counter-examples to these observations.

8.11 Summary

The static analysis program produced accurately manages to determine safety over a range of typical pointer programs. During the analysis real safety concerns were identified, and amended versions were produced.

Chapter 9:

Conclusions and Further Work

This chapter compares how the program matches the aims of the project, and the tasks for which the program could be used in its existing form. Some future possible enhancements are also covered.

9.1 Existing Performance

The program as it currently is meets many of the initial objectives set out in the aims of the project. A working analyser has been produced which can generate a proof that a procedure is total – both that it does not crash and that it does terminate.

The one deviation from the initial aims of the project was the introduction of the `acyclic` keyword. An unmodified Pasta program is unlikely to validate correctly, however the additional annotation is a relatively small change if the data structure is well understood.

If a graph analyser was created, as is the intention, then when combined with this program guarantees of termination, safety and correctness could be given. Using the associated Pasta to C convertor this proven code could then be used in general purpose programs. This would provide very high levels of safety, in a programming language that is used by many people, where safety is usually much harder to prove.

The program provides more assurances than tools such as Splint [14], while requiring less effort to apply compared to approaches such as B [2]. Other tools such as SPARK [7] avoid aliasing, but this analysis handles it in an automated manner, without being overly pessimistic. Uniquely for most static analysis tools, proof of termination is given, with only minimal user input.

9.2 Compiler Optimisations

One use for the analysis program, other than the clear benefit of having safer code, would be for use in an optimising compiler. For many programs, the cost of a runtime exception occurring is quite high. For example, using the GCC [24] compiler, turning on C++ exception handling can cause the resulting program to become up to 9 times larger. If it could be shown that no exceptions occurred, even just within certain procedures, extra checks could be avoided.

For the statement `x = y->field` it is possible that if this code is safe, `y` could still be possibly more than one subtype. This means that at runtime, the actual memory address of `field` will have to be computed by looking at the subtype of `y`. If the subtype could be precisely determined, this extra level of indirection could be avoided.

This last use would probably be more suited to forward analysis, as that would require only one pass to determine information about all variables in the program.

9.3 Better Error Reporting

The tool currently uses disjunctive normal form for all predicates, and this is also the representation used to display preconditions to the user. For a human, a more natural form would include additional operators such as $\Rightarrow$ and $\Leftrightarrow$. In addition, where the program expands out brackets to achieve the standard representation, a human reader would probably find fewer terms easier to comprehend and work with.

9.4 Removal of Annotations

Currently the Pasta programs require some additional annotations in order to be proved total. A worthy goal would be trying to reduce these annotations in number, so that more programs worked without modification in any way. The most obvious annotation to remove would be the `acyclic` notation, which could be automatically computed in some circumstances.

Other annotations on the code relate to the disjoint subtypes, and by extending the language in this way, programmers could write their intentions more directly. This would also allow more analysis to be performed statically in the type checking phase, hopefully discovering more errors at an earlier stage.

9.5 Extended Language

The Pasta language as it currently stands is relatively simple compared to languages such as C and Ada. Many constructs are missing, but one particular feature lacking is the concept of arrays. Also there are no mathematical operations on integers, which would be useful for writing many programs.

As a result of these intentional omissions, various classes of errors are avoided. If the Pasta language was extended to include these concepts, or an alternative language such as Ada was analysed, then additional features would be required in addition to those currently in the analyser.

One interesting area for investigation would be whether the predicate solver and `update`/`oblig` pattern could be preserved, or whether this would be insufficient. The current implementation model is particular concise when determining the properties of statements, and this would be useful to maintain.

9.6 Forward Analysis

Forward analysis was both designed and implemented, and produced a fairly powerful analyser. While it did have various limitations, by investigating other design options it is possible that these may be overcome. In addition, forward analysis has some applications available to it which backward analysis lacks.

Finally, forward analysis allows the abstract state to be viewed after each statement. While this is not always very helpful for fixing potential errors, it could be very useful for people trying to learn how a particular procedure works.

9.7 Performance Improvements

The issue of performance has not been discussed in this document, and all procedures are implemented as direct mathematical translations where possible, with no thought to speed. As a result, execution times for short procedures can be in the range of minutes. In order to create a tool that would be useable on large projects, a speedup would be required.

The first place to look for an increase in performance would be the predicate solver. From experiments it appears that almost all the execution time is spent simplifying and rearranging formulae. The representation used for the terms is linked lists, and many inefficient algorithms are applied to them. One example is the simplification using the idempotence property $p = p \wedge p$, which is $O(n^2)$ where n is the length of the list. Simple algorithms exist which achieve $O(n \log n)$, and by optimising further, this cost could be avoided every time a term is simplified.

9.8 Procedure Isolation

The current design of the program expands all procedures inline to validate the `main` procedure. This procedures a very large body of code, and in a large system, this is very likely to be infeasible. If each

procedure could be analysed in isolation, with the preconditions then used afterwards the system could then be designed to scale linearly with the number of procedures.

Bibliography

[1] ISO/IEC 13568:2002. Information technology—Z formal specification notation—syntax, type system and semantics. International Standard.

[2] Jean-Raymond Abrial. *The B-Book – Assigning Programs to Meanings*. Cambridge University Press, December 1996.

[3] Hiralal Agrawal, Joseph R. Horgan, Edward W. Krauser, and Saul A. London. Incremental regression testing. In *Conference on Software Maintenance*, pages 348–357, 1993.

[4] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Military Standard Ada Programming Language*, February 17 1983. Also MIL-STD-1815A.

[5] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[6] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *Programming Languages - C++ ISO/IEC 14882:1998(E)*, September 01 1998.

[7] John Barnes. *High Integrity Software: The SPARK Approach*. Addison Wesley, 2003.

[8] Roman Bartak. Constraint Programming: In Pursuit of the Holy Grail. *Week of Doctoral Students*, pages 555–564, June 1999.

[9] David R Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *ACM SIGPLAN'90 Conference on Programmaing Language Design and Implementation*, Jun 1990.

[10] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference of Functional Programming*, 2000.

[11] Microsoft Corporation. *Microsoft Visual Basic 6.0 Reference Library*. Microsoft Press, August 1998.

[12] Coverity. SWAT. `http://www.coverity.com/`, 2004.

[13] Andy Chou et. al. Stanford Checker. `http://marc.theaimsgroup.com/?l=linux-kernel&m=104155431311370&w=2`, 2003.

[14] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, Jan/Feb 2002.

[15] M. Gerlek, M. Wolfe, and E. Stoltz. A reference chain approach for live variables, 1994.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, second edition, 2000.

[17] Anthony Hall and Roderick Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, pages 18–25, Jan/Feb 2002.

[18] International Standards Organisation. *Pascal ISO 7185*, 1990.

[19] S. C. Johnson. Lint, a c program checker. Technical report, Bell Laboratories, July 1978.

[20] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, first edition, 2003.

[21] William Landi and Barbara G. Ryder. A Safe Approximation Algorithm for Interprocedural Pointer Aliasing. *ACM SIGPLAN '92*, pages 235–248, 1992.

[22] John C Martin. *Introduction to Languages and the Theory of Computation*. McGraw Hill, second edition, 1997.

[23] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[24] GNU Project. GNU Compiler Collection. `http://gcc.gnu.org/`, 2004.

[25] Colin Runciman. Pasta Shell: Revision and First Implementation. Internal email memorandum, September 2002.

[26] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24:217–298, May 2002.

[27] Alan Mathinson Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.

Appendix A:

Expanded List `insert` Procedure

This appendix gives a detailed translation of the linked list insertion procedure, both in standard Pasta and the reduced version of Pasta. Details on the translation process can be found in section .

A.1 Standard Pasta

```
insert(int i, ptr s) {
    while (s::cons && s->head < i) s = s->tail;
    if (s::nil || s->head > i) *s = *cons(i,copy(s));
}
```

A.2 Reduced Pasta

```
insert(int i, ptr s) {
    if (s::cons) {
        t2 = s->head;
        if (t2 < i) {
            t1 = 1;
            while (t1 == 1) {
                s = s->tail;
                if (s::cons) {
                    t3 = s->head;
                    if (t3 < i)
                        t1 = 1;
                    else
                        t1 = 0;
                } else
                    t1 = 0;
            }
        }
    }
    if (s::nil) {
        t4 = nil();
        if (s::nil) {
            t5 = nil();
            *t4 = *t5;
        } else {
            t6 = cons();
            t7 = s->head;
            t6->head = t7;
            t8 = s->tail;
            t6->tail = t8;
            *t4 = *t6;
        }
        t9 = cons();
        t9->head = i;
        t9->tail = t4;

        *s = *t9;
    } else {
        t10 = s->head;
        if (t10 > i) {
            t11 = nil();
            if (s::nil) {
                t12 = nil();
                *t11 = *t12;
            } else {
                t13 = cons();
                t14 = s->head;
                t13->head = t14;
                t15 = s->tail;
                t13->tail = t15;
                *t11 = *t13;
            }

            t16 = cons();
            t16->head = i;
            t16->tail = t11;

            *s = *t16;
        }
    }
}
```

Appendix B:

Queue Sample

This is the original code for the queue, with the safe modification in comments.

```
-- version 2.0

queue acyclic(tl) acyclic(front,rear) {
    nil();
    cons(int hd, ptr tl);
    queue(ptr front, ptr rear);
}

makeq(ptr q) {
    ptr d = nil();
    *q = *queue(d, d);

--To cause a failure in the original version
--just uncomment the following line
-- q->rear = q;
}

enq(int i, ptr q) {
--Modified version
--  ptr rear = q->rear;
--  q->rear = copy(q->rear);
--  *rear = *cons(i,q->rear);

--Original version
        *q->rear = *cons(i,copy(q->rear));
        q->rear = q->rear->tl;
}

deq(ptr q) {
    if (q->front::cons) q->front = q->front->tl;
}

main() {
    ptr qe = nil();
    makeq(qe);
    enq(1,qe); enq(2,qe);
    deq(qe); deq(qe); enq(0, qe);
}
```

Appendix C:

Thread Tree Sample

This is the original code for the threaded tree, without any modifications.

```
thread {
       leaf();
       branch(ptr prev, ptr left, int n, ptr right, ptr next);
       root(ptr first, ptr top, ptr last);
}

maket(ptr t) {
       ptr l = leaf();
       *t = *root(l,l,l);
       t->first = t;
       t->last = t;
}

insert(int i, ptr t) {
       if (t->top::leaf)
       {
              t->first = t->top;
              t->last = t->top;
              *t->top = *branch(t,leaf(),i,leaf(),t);
       }
       else
       {
              t = t->top;
              while (t->n < i && t->right::branch ||
                        t->n > i && t->left::branch)
              {
                     if (t->n < i) t = t->right;
                     else t = t->left;
              }
              if (t->n < i)
              {
                     *t->right = *branch(t,leaf(),i,leaf(),t->next);
                     if (t->next::branch)
                            t->next->prev = t->right;
                     else
                            t->next->last = t->right;
                     t->next = t->right;
              }
              else if (t->n > i)
              {
                     *t->left = *branch(t->prev,leaf(),i,leaf(),t);
                     if (t->prev::branch)
                            t->prev->next = t->left;
                     else
                            t->prev->first = t->left;
```

```
                        t->prev = t->left;
                }
        }
}

delete(int i, ptr t) {
        t = t->top;
        while (t::branch && t->n != i)
        {
                if (t->n < i)
                        t = t->right;
                else
                        t = t->left;
        }
        if (t::branch)
        {
                while ((t->left::branch || t->right::branch) &&
                                                t->next::branch)
                {
                        if (t->right::branch)
                        {
                                t->n = t->next->n;
                                t->next->n = i;
                                t = t->next;
                        }
                        else
                        {
                                t->n = t->prev->n;
                                t->prev->n = i;
                                t = t->prev;
                        }
                }
                if (t->prev::branch)
                        t->prev->next = t->next;
                else
                        t->prev->first = t->next;
                if (t->next::branch)
                        t->next->prev = t->prev;
                else
                        t->next->last = t->prev;
                *t = *leaf();
        }
}


main() {
        ptr t = leaf();
        maket(t);
        insert(2,t); insert(1,t); insert(3,t);
        delete(2,t); delete(1,t);
}
```

Appendix D:

Regression Tests

The following are regression tests that were used to check the program during development. The tests that start with `fail` are instances where the analysis program should not be able to prove the preconditions.

```
-- Penne Test Suite
-- Test's for the Penne validation program

struct acyclic(q) {
       nil();
       cons(int n, ptr p, ptr q);
}

main() {}

-- BASIC TESTS
---------------
-- basic application of the loop structures and very simple
   information flow

testempty(ptr p) {
}

testif(ptr p) {
       if (p::cons)
              p = p->q;
}

testwhile(ptr p) {
       while (p::cons)
              p = p->q;
}

testelse(ptr p) {
       if (p::nil)
              p = p;
       else
              p = p->q;
}

-- COMPOUND TESTS
------------------
-- tests where there are more than one conditional variable
testifand(ptr p) {
       if (p::cons && p->n < 12)
              p = p->q;
}

testwhileand(ptr p) {
       while (p::cons && p->n < 12)
              p = p->q;
}

-- ASSIGNMENT TESTS
--------------------
-- requires knowledge about assignment to be propogated
assignvar(ptr r) {
       if (r::cons) {
              ptr p = r;
              p = p->p;
       }
}

assignctor(ptr q) {
       ptr p = cons(0, q, q);
       p = p->p;
}
```

```
assignctordeep(ptr q) {
        if (q::cons) {
                ptr p = cons(0, q, q);
                p = p->p->p;
        }
}

assignvardeep(ptr q) {
        if (q::cons) {
                ptr p = cons(0, q, q);
                p->p = cons(0, q, q);
                p = p->p->p;
        }
}

assignstarctor(ptr q) {
        *q = *cons(0, q, q);
        q = q->p;
}

assigntrick(ptr q)
{
        q = cons(0, q, q);
        *q->p = *cons(0, q, q);

        --only true because if q was overwritten, then q is still a
            cons
        q = q->p;
}


-- TRICHOTOMY TESTS
--------------------
-- require trichotomy information
trichotomy(int n, ptr q)
{
        if (n > 0)
        {
                if (n <= 0)
                        n = q->n;
        }
}


-- CRASH TESTS
---------------
-- all of these resulted in a crash in the analysis program at some
    point
testbasic(ptr p) {
        if (p::cons)
                p = p->q;
        while (p::cons)
                p = p->q;
        if (p::nil)
                p = p;
        else
                p = p->q;
}


-- FAIL TESTS
--------------
-- all of these SHOULD fail, or the evaluator is wrong...
faila(ptr p) {
        ptr x = p->p;
}

faildead(ptr p) {
        while (p::cons && p::nil)
        { }
}

failinf(ptr p) {
        while (p::cons || p::nil)
        { }
}
```

Appendix E:

Source Code

The following source code comes from the backward and forward analysis programs. Only selected modules are included here, those which perform the complex and interesting sections.

E.1 Statement Abstraction

This module performs abstraction on the Pasta language. While this module is used for the backward analysis, a very similar module is present in the forward analysis.

```
module Abstract(Abstract(..), abstract) where

import Maybe
import Gen
import "../kernel/Syntax.hs"
import "../kernel/ValueType.hs"
import SyntaxEx


data Abstract = Loop Int Expr [Abstract]
              | Choice Expr [Abstract] [Abstract]
              | Assign Location Location
              | Star Location Location
              | Number Location Int
              | New Location Name
              deriving(Eq)


instance Show Abstract where
    -- with discarded children
    show (Loop i cond body) = "Loop:" ++ show i ++ " ("
         ++ showExpr cond ++ ")"
    show (Choice cond _ _) = "Both (" ++ showExpr cond
         ++ ")"

    -- others
    show (Assign x y) = showLoc x ++ " = " ++ showLoc y
    show (Star x y) = showLoc x ++ " *=* " ++ showLoc y
    show (Number x n) = showLoc x ++ " = " ++ show n
    show (New x name) = showLoc x ++ " = new(" ++ name
         ++ ")"

    -- show everything, with nice indenting
    showList = showString . unlines . g
        where
            f :: Abstract -> [String]
            f (Choice cond true false) =
                ["if (" ++ showExpr cond ++ ") {"] ++
                g true ++
                if null false then ["}"] else
                    ["} else {"] ++
                    g false ++
                    ["}"]

            f (Loop i cond body) = ["loop:" ++ show i ++ "
                 (" ++ showExpr cond ++ ") {"] ++ g body
                 ++ ["}"]
            f x = [show x]

            g :: [Abstract] -> [String]
            g x = map (" " ++) (concat (map f x))
```

```
abstract :: Program -> Operation -> [Abstract]

abstract prog op = abss prog (opBody op) 0


type VarRename = (Name -> Name)


abss :: Program -> Command -> Int -> [Abstract]

abss prog (While x y) d = [Loop d x (abss prog y (d+1))]

abss prog (If x t f) d = [Choice x (abss prog t (d+1)) f']
    where f' = if isJust f then abss prog (fromJust f) (d
        +1) else []

abss prog (Block decl comm) d = concat (map f decl) ++
    concat (map g comm)
    where
        toRen = map fieldName (concat (map declFields
            decl))
        varRen x = if x `elem` toRen then x ++ show d else
             x

        f (Decl flds exps) = pllassign prog (map (Loc
             . (:[]) . varRen . fieldName) flds) exps
        g x = abss prog (renComm varRen x) (d+1)


abss prog (Call name xs) d = abss prog (Block [Decl params
     xs] [body]) d
    where (Operation (Struct _ params) body) = fromJust (
        findOp prog name)

abss prog (x := y) d = pllassign prog x y

abss prog stmt d = error ("Ziti.Abstract.abs, unrecognised
     statement: " ++ show stmt)


pllassign :: Program -> [Expr] -> [Expr] -> [Abstract]

--optimize the common case (also reduces output complexity)
pllassign prog [Loc x] [y] = assign prog x y
pllassign prog [Str (Loc x)] [y] = assign prog x y

pllassign prog locs exps = concat a ++ b
    where
        (a, b) = unzip (zipWith f locs exps)

        f (Str loc) (Str ex) = g Star loc ex
        f loc ex = g Assign loc ex

        g ctor (Loc loc) ex = (assign prog loc' ex, ctor
            loc loc')
            where loc' = addPre "_" loc


assign :: Program -> Location -> Expr -> [Abstract]

assign prog loc (Loc e) = [Assign loc e]

assign prog loc (Str e) = assign prog loc' e ++ [Star loc
    loc']
    where loc' = addPre "_" loc

assign prog loc (Con name es) = New loc name : concat (
    zipWith f allFields es)
    where
        allFields = map fieldName (structFields (
            getStruct prog name))
```

```
        f l e = assign prog (loc ++ [l]) e

assign prog loc (Cpy e) = stmts ++ f (sigStructs (progSig
    prog))
    where
        (loc', stmts) = getLoc prog (addPre "_" loc) e

        f [x] = g x
        f (x:xs) = [Choice (Kin loc' (structName x)) (g x)
             (f xs)]

        g x = New loc (structName x) : map (h . fieldName
            ) (structFields x)
            where h x = Assign (loc ++ [x]) (loc' ++ [x])

assign prog loc (Val (N num)) = [Number loc num]

assign prog loc expr = error ("Ziti.Abstract.assign,
    unrecognised assignment: " ++ show loc ++ " := " ++
    show expr)


getLoc :: Program -> Location -> Expr -> (Location, [
    Abstract])
getLoc prog loc (Loc x) = (x, [])
getLoc prog loc expr = (loc, assign prog loc expr)



addPre :: String -> Location -> Location
addPre pre loc = [pre ++ joinMid "-" loc]
```

```
renLoc :: VarRename -> Location -> Location
renLoc v (l:loc) = v l : loc


renExprs v xs = map (renExpr v) xs

renExpr :: VarRename -> Expr -> Expr
renExpr v (Loc loc ) = Loc (renLoc v loc)
renExpr v (Con n xs ) = Con n (map (renExpr v) xs)
renExpr v (Cpy x ) = Cpy (renExpr v x)
renExpr v (Val val ) = Val val
renExpr v (Str x ) = Str (renExpr v x)
renExpr v (Rel x r y) = Rel (renExpr v x) r (renExpr v y)
renExpr v (Kin loc n) = Kin (renLoc v loc) n
renExpr v (Or xs ) = Or (renExprs v xs)
renExpr v (And xs ) = And (renExprs v xs)


renComm :: VarRename -> Command -> Command
renComm v (x := y) = renExprs v x := renExprs v y
renComm v (If a x y) = If (renExpr v a) (renComm v x) (if
    isNothing y then Nothing else Just (renComm v (
    fromJust y)))
renComm v (While a x) = While (renExpr v a) (renComm v x)
renComm v (Call name xs) = Call name (renExprs v xs)
renComm v (Block dec com) = Block (map f dec) (map (
    renComm v) com)
    where f (Decl flds exps) = Decl flds (renExprs v exps
        )
```

E.2 Predicate Engine

This module manages the predicate logic statements, including simplifying and displaying them. It is required by the backward analysis program only.

```
module Proof where

import Gen
import List
import Maybe


data Proof x = Pass String
             | Fail String
             | PAnd [Proof x]
             | POr [Proof x]
             | Imp (Proof x) (Proof x)
             | Not (Proof x)
             | Proof x
             deriving(Eq, Ord, Show)


fromProof (Proof x) = x


isLogic :: Proof x -> Bool
isLogic (Pass _ ) = True
isLogic (Fail _ ) = True
isLogic (PAnd _ ) = True
isLogic (POr _ ) = True
isLogic (Not _ ) = True
isLogic (Imp _ _) = True
isLogic _ = False

isPass :: Proof x -> Bool
isPass (Pass _) = True
isPass _ = False

isFail (Fail _) = True
isFail _ = False

isOr (POr _) = True
isOr _ = False

isNot (Not _) = True
isNot _ = False

deNot (Not x) = x

mapProof :: (Proof x -> Proof x) -> Proof x -> Proof x
mapProof f (Pass s) = Pass s
mapProof f (Fail s) = Fail s
mapProof f (PAnd x) = f (PAnd (mapProofs f x))
mapProof f (POr x) = f (POr (mapProofs f x))
mapProof f (Not x) = f (Not (mapProof f x))
mapProof f (Imp x y) = f (Imp (mapProof f x) (mapProof f y
    ))
mapProof f x = f x

mapProofs :: (Proof x -> Proof x) -> [Proof x] -> [Proof x
    ]
mapProofs f = map (mapProof f)

mapValue :: (x -> Proof x) -> Proof x -> Proof x
mapValue f x = mapProof g x
    where g x = if isLogic x then x else f (fromProof x)


getValue :: Proof x -> [x]
getValue (Proof x) = [x]
getValue (PAnd x) = getValues x
getValue (POr x) = getValues x
getValue (Not x) = getValue x
getValue (Imp x y) = getValue x ++ getValue y
getValue _ = []

getValues = concat . map getValue


simplify :: (Show x, Ord x) => Proof x -> Proof x
-- version 2 - full disjunctive normal form Or(And(Not(x)))
simplify x = (reduce . reduceAdv . reduce . toDnf . reduce
```

```
      . remImp) x


reduceAdv :: (Show x, Ord x) => Proof x -> Proof x
reduceAdv x = mapProof f x
    where
        -- a v (a ^ c) -> a
        -- a v a -> True
        f (POr xs) = if isJust res then Pass (fromJust res)
             else POr (filterBy f xs)
            where
                res = negDupe xs
                f x = all (\y -> not (y `psubset` x)) xs

        -- a ^ a ^ b -> False
        f (PAnd xs) = if isJust res then Fail (fromJust res
            ) else PAnd xs
            where res = negDupe xs

        f x = x


        negDupe xs = if null res then Nothing else Just (
            show (head res))
            where
                res = intersect (map deNot nots) just
                (nots, just) = partition isNot xs


        psubset (PAnd x) (PAnd y) = x `subset` y && length
             x < length y
        psubset x (PAnd y) = [x] `subset` y && lenGt1 y
        psubset x y = False


toDnf :: (Show x, Ord x) => Proof x -> Proof x
toDnf x = mapProof (f . reduce) (downNot x)
    where
        -- move all Not expressions to their lowest level
        downNot (Not (Not x)) = x
        downNot (Not (PAnd x)) = POr (map (downNot . Not)
            x)
        downNot (Not (POr x)) = PAnd (map (downNot . Not)
            x)
        downNot (PAnd x) = PAnd (map downNot x)
        downNot (POr x) = POr (map downNot x)
        downNot x = x

        f (PAnd x) = g ors ands -- errorS (POr (g ors [PAnd
            ands])) --if null ors then PAnd ands else toDnf (POr
             (g ors [PAnd ands]))
            where
                (ors, ands) = partition isOr (hoist f x)

                g [] ands = PAnd (hoist f ands)
                g (POr o:ors) ands = (toDnf . reduce) (POr
                     (map (\x -> PAnd [x,res]) o))
                    where res = g ors ands

                f (PAnd x) = Just x
                f _ = Nothing

        f (POr x) = POr (hoist f x)
            where
                f (POr x) = Just x
                f _ = Nothing

        f x = x


-- basic simplification techniques
reduce :: (Show x, Ord x) => Proof x -> Proof x
reduce x = mapProof f x
    where
        f (Not (Not x)) = x
        f (PAnd xs) = simpAndOr xs True
        f (POr xs) = simpAndOr xs False
        f (Not (Pass s)) = Fail s
```

```
        f (Not (Fail s)) = Pass s
        f x = x


remImp :: Proof x -> Proof x
remImp x = mapProof f x
    where
        f (Imp x y) = POr [Not x, y]
        f x = x



simpAndOr xs isAnd =
        if null xs then (if isAnd then Pass else Fail) "-
            none-"
        else if not (null oneL) then head oneL
        else if null manyL then head res
        else if lenEq1 manyL then head manyL
        else (if isAnd then PAnd else POr) manyL
    where
        oneL = filter one res
        manyL = filter (not . many) res
        res = sortUnique (hoist (h isAnd) xs)

        h True (PAnd xs) = Just xs
        h False (POr xs) = Just xs
        h _ _ = Nothing

        many = isAndPass isAnd
        one = isAndPass (not isAnd)

        isAndPass True (Pass _) = True
        isAndPass False (Fail _) = True
        isAndPass _ _ = False
```

```
hoist :: (x -> Maybe [x]) -> [x] -> [x]
hoist f [] = []
hoist f (x:xs) = (if isJust res then (++) (fromJust res)
    else (:) x) (hoist f xs)
    where res = f x
```


E.3 Backward Execution Engine

This module analyses the actual statements, including the `update` and `oblig` routines. It is the main logic behind the backward analysis program.

```
module Ziti(ziti) where

import "../kernel/Syntax.hs"
import "../kernel/ValueType.hs"
import Proof
import Abstract
import List
import SyntaxEx
import Maybe
import Gen
```

```
import Debug
import Annot



-- DRIVER
----------
ziti :: Program -> [Abstract] -> [Annot]
ziti prog stmt = execl prog stmt (Pass "Initial condition"
    )
```

```
-- a list of annotationes must have an obligation at either end
execl :: Program -> [Abstract] -> Proof0 -> [Annot]
execl prog stmt proof = f (reverse stmt) [Annot proof]
    where
        f [] ann = ann
        f (x:xs) ann = f xs (res ++ ann)
            where res = exec prog x (preCond ann)


exec :: Program -> Abstract -> Proof0 -> [Annot]
exec prog (Choice cond true false) proof = [Annot (prune (
    PAnd [trueC, falseC])), x]
    where
        x = AnnotIf cond (Annot trueC:trueB) (Annot falseC
            :falseB)
        trueB = execl prog true proof
        falseB = execl prog false proof
        trueC = prune (boolExpr prog cond (preCond trueB))
        falseC = prune (boolExpr prog (notExpr cond) (
            preCond falseB))


exec prog (Loop id cond body) proof =
        [
            Annot (if isNothing res then Fail "Loop does
                not fixpoint" else prune (PAnd fixp)),
            (if isNothing res
                then AnnotLp cond Nothing (Right (take 3
                    proofs))
                else AnnotLp cond fwdVar (Left (last fixp))
            )
            (head annots)
        ]
    where
        -- the value after the loop has done the termination
        proof' = prune (trueExpr prog (notExpr cond) proof
            )
Just fixp = res
res = fix (take 3 proofs)

proofs = proof' : map (prune . mapValue doneLoop
     . preCond) annots

annots = f (prune (PAnd [proof', loopCond]))
    where f x = res : f (preCond res)
            where res = eval x

eval :: Proof0 -> [Annot]
eval p = Annot r1 : Annot r2 : res
    where
        r1 = if isAnyLoop `elemBy` getValue r2 then
            r1t else Fail "No forward motion"
        r1t = prune (boolExpr prog cond r2)
        r2 = prune (mapValue reLoop (preCond res))
        res = execl prog body p


fwdVar = ifNothing res (loc, names)
    where
        (Just (FwdLoop _ _ loc names)) = res
        res = find isAnyLoop (getValue (preCond (
            annots !! (length fixp - 1))))

-- all the possible variables which could be used
loopCond = POr (Fail "No forward motion" : map
    Proof (filter isAnyLoop (getValue (preCond (eval
     (PAnd [proof', loopCondPre]))))))
loopCondPre = POr [fn var|fn<-newLoops, var<-
    posVars]
posVars = (nub . concat) (map (tail . inits) (
    getLocAbstracts body))
newLoops = map (\x y -> Proof (FwdLoop id y y (
    propAcyclic x))) (sigProps (progSig prog))
```

```
        -- auxiliary functions for testing the FwdLoop conditions
        isAnyLoop (FwdLoop i l2 l1 names) = i == id
        isAnyLoop _ = False

        doneLoop x = if isAnyLoop x then Pass "FwdLoop"
            else Proof x

        reLoop s@(FwdLoop i l2 l1 names) | i == id = if (l1
             == init l2) && (last l2 `elem` names) then
            Proof (FwdLoop i l1 l1 names) else Fail (show
            s)
        reLoop x = Proof x



exec prog stmt proof = [Annot (prune proof'), AnnotAb stmt
    ]
    where proof' = PAnd [oblig prog stmt, mapValue (
        update prog stmt) proof]


-- conditions that must hold as a result of an expression
condExpr :: Expr -> Proof0
condExpr (Kin x name) = Proof (Test x name)
condExpr (NKin x name) = Not (condExpr (Kin x name))

condExpr (Rel (Loc x) op (Loc y)) = Proof (Relation x op y
    )
condExpr (Rel (Loc x) op (Val (N y))) = Proof (RelNum x op
     y)
condExpr (Rel (Val (N x)) op (Loc y)) = Proof (RelNum y (
    notRel op) x)


-- use De Morgans law to make not the lowest term
notExpr :: Expr -> Expr
notExpr (And x) = Or (map notExpr x)
notExpr (Or x) = And (map notExpr x)
notExpr (Kin x name) = NKin x name
notExpr (Rel x op y) = Rel x (notRel op) y
notExpr (NKin x name) = Kin x name


--Compound expression, includes && or ||
compExpr :: (Expr -> Proof0 -> Proof0) -> Expr -> Proof0
     -> Proof0
compExpr f (And [x] ) proof = compExpr f x proof
compExpr f (Or [x] ) proof = compExpr f x proof

compExpr f (And (x:xs)) proof = compExpr f x (compExpr f (
    And xs) proof)
compExpr f (Or (x:xs)) proof =
    PAnd [compExpr f x proof,
          compExpr f (notExpr x) (compExpr f (Or xs) proof)
              ]

compExpr f cond proof = f cond proof


boolExpr :: Program -> Expr -> Proof0 -> Proof0
boolExpr prog = compExpr f
    where f cond proof = PAnd [obExpr prog cond, Imp (
        condExpr cond) (mapValue (updateExpr cond) proof)]

trueExpr :: Program -> Expr -> Proof0 -> Proof0
trueExpr prog = compExpr f
    where f cond proof = PAnd [obExpr prog cond, mapValue
         (updateExpr cond) proof]



-- OBLIGATIONS
---------------

oblig :: Program -> Abstract -> Proof0

oblig prog (Assign l1 l2) = PAnd [obLoc prog l1, obLoc
```

```
    prog l2]
oblig prog (Star l1 l2) = PAnd [obLoc prog l1, obLoc prog
    l2]
oblig prog (Number loc _) = obLoc prog loc
oblig prog (New loc _) = obLoc prog loc
oblig prog x = error ("Unhandled oblig for " ++ show x)

obExpr :: Program -> Expr -> Proof0
obExpr prog x = PAnd (map (obLoc prog) (getLocExpr x))

obLoc :: Program -> Location -> Proof0
obLoc prog loc = PAnd (map f (inits loc))
    where
        f [] = Pass ""
        f [_] = Pass ""
        f loc = obSelector prog (init loc) (last loc)

obSelector :: Program -> Location -> Name -> Proof0
obSelector prog var sel = Not (POr (f false)) -- POr (Not
     (POr (f false)) : f true)
    where
        f xs = map (Proof . Test var) xs
        (true, false) = partition (\x -> sel `elem`
            typeSelectors prog x) (progTypeNames prog)



-- UPDATES
------------

update :: Program -> Abstract -> Oblig -> Proof0


update prog (New l n) p = case p of
    s@(Test l' n') -> if l == l' then porf (n == n') (show
         s) else Proof p
    s@(Alias x y) | x == y -> Pass (show s)
    s@(Alias x y) | x == l || y == l -> Fail (show s)
    x -> Proof x

update prog (Number l v) p = case p of
    RelNum l' op v' | l == l' -> porf (runRel v op v') (
        show "RelNum " ++ show v ++ " " ++ show op ++ "
        " ++ show v')
    Relation l1 op l2 | l == l2 -> Proof (RelNum l1 op v)
    Relation l1 op l2 | l == l1 -> Proof (RelNum l2 (
        notRel op) v)
    _ -> Proof p

update prog (Assign l1 l2) p = case p of
    x -> Proof (mapLocOblig (renLoc l1 l2) p)


update prog (Star l1 l2) p = case p of
    x -> f (nub (getLocOblig p)) (Proof p)
        where
            f :: [Location] -> Proof0 -> Proof0
            f [ ] p = p
            f (x:xs) p = PAnd [Imp (Proof (Alias x l1)) (f
                xs ren), Imp (Not (Proof (Alias x l1))) (f
                xs p)]
                where ren = mapValue (Proof . mapLocOblig (
                    renLoc x l2)) p

update prog abss ob = error ("Unhandled update (" ++ show
     abss ++ ") (" ++ show ob ++ ")")


updateExpr :: Expr -> Oblig -> Proof0

updateExpr (NKin l n) p = case p of
    s@(Test l' n') -> if l == l' then porf (n /= n') (show
         s) else Proof p
    _ -> Proof p

updateExpr ( Kin l n) p = case p of
    s@(Test l' n') -> if l == l' then porf (n == n') (show
         s) else Proof p
```

```
    _ -> Proof p

{-
updateExpr (Rel x op (Num y)) p = case p of
    s@(RelNum x' op' n') | x == x' && op' == op -> Pass (show
        s)
    s@(RelNum x' op' n') | x == x' && op' == notRel op -> Fail
        (show s)
    _ -> Proof p
-}

updateExpr (Rel x op y) p = case p of
    _ -> Proof p


updateExpr expr ob = error ("Unhandled updateExpr (" ++
    show expr ++ ") (" ++ show ob ++ ")")


porf True x = Pass x
porf False x = Fail x


-- HELPER
----------


renLoc :: Location -> Location -> Location -> Location
renLoc from to val = if take lenFrom val == from then to
     ++ drop lenFrom val else val
    where lenFrom = length from


getLocAbstracts = concat . map getLocAbstract

getLocAbstract :: Abstract -> [Location]
{-
getLocAbstract (Check loc name) = [loc]
getLocAbstract (Know loc name) = [loc]
-}
getLocAbstract (Loop id expr abss) = getLocExpr expr ++
    getLocAbstracts abss
getLocAbstract (Choice expr abs1 abs2) = getLocExpr expr
     ++ getLocAbstracts abs1 ++ getLocAbstracts abs2
getLocAbstract (Assign l1 l2) = [l1, l2]
getLocAbstract (Star l1 l2) = [l1, l2]
getLocAbstract (Number loc id) = [loc]
getLocAbstract (New loc name) = [loc]


prune :: Proof0 -> Proof0
prune x = simplify (mapValue f x)
    where
        f a@(Alias x y) | x == y = Pass (show a)
        f x = Proof x
```


E.4 Forward Execution Engine

This module analyses the actual statements, and maps the abstract state over each statement. It is the main logic behind the forward analysis program.

```
module Check(check, Error(..)) where

import Know
import Abstract
import "general/Gen"
import Opts
import "../kernel/Syntax"
import List
```

```
import Maybe
import SyntaxEx

data Error = Unsafe Location Know
           | Safe Location
           | MayLoop Abstract
           | DefLoop Abstract
           | NoLoop Abstract (Maybe Location)
           | Dead Abstract Know
           | FixFail Abstract
           | Debug Abstract Know (Maybe Know)
           | Return Know
           | Comment String

instance Show Error where
    show (Unsafe name k) = "UNSAFE: " ++ showLoc name ++
        " # " ++ show k
    show (Safe name ) = "safe: " ++ showLoc name
    show (MayLoop a ) = "MAYLOOP: " ++ show a
    show (DefLoop a ) = "DEFLOOP: " ++ show a
    show (NoLoop a name) = "noloop: " ++ show a ++ " "
         ++ (if isJust name then showLoc (fromJust name)
        else "CAN'T loop")
    show (Dead a k ) = "DEAD: " ++ show a ++ " " ++ show
         k
    show (FixFail a ) = "FIXFAIL: " ++ show a
    show (Debug a k1 k2) = "debug: " ++ show a ++ " | <"
         ++ show k1 ++ "> = <" ++ showMay k2 ++ ">"
    show (Return k ) = "return: " ++ show k
    show (Comment s ) = "-- " ++ s

showMay Nothing = "<nothing>"
showMay (Just x) = show x


isSafe :: Error -> Bool
isSafe (Safe _ ) = True
isSafe (NoLoop _ _) = True
isSafe _ = False

isDebug :: Error -> Bool
isDebug (Debug _ _ _) = True
isDebug _ = False


-- this state is maintained at every chk function position
-- Maybe Know is Nothing if you can't reach this point (break just
    before), or Just x if valid knowledge
-- Maybe Know is the break knowledge, if available
-- [[Location]] is a list of variables with forward momentum, which
    haven't been destroyed
-- head Location is the list of the current loop
-- tail Location is the hierarchical loops
-- if Maybe then break was encountered, hence no forward required
-- [Error] is a list of errors that have been generated
type ChkState = (Maybe Know, Maybe Know, [Maybe [
    Location]], [Error])


check :: Signature -> [Abstract] -> [Error]
check sig xs = filter f errs ++ [Return know]
    where (Just know, Nothing, _, errs) = chkList (new
        sig) [Just []] xs
          f x = (isSafe x ==> optAllTests) && (isDebug x
               ==> optTrace)


-- return what you know, what you broke out knowing, errors
    generated
chkList :: Know -> [Maybe [Location]] -> [Abstract] ->
    ChkState

-- simple cases
-- break on its own asserts break must be the last entry in a list
chkList know vars [Break] = (Nothing, Just know, Nothing
     : tail vars, [])
chkList know vars [x ] = annotate know vars x
chkList know vars [ ] = (Just know, Nothing, vars, [])
```

```
-- and more complex
chkList know vars (x:xs) = (k2, joinMKnow b1 b2, v2, e1++
    e2)
    where
        (Just k1, b1, v1, e1) = annotate know vars x
        ( k2, b2, v2, e2) = chkList k1 v1 xs


-- add debug annotations around the know statement
annotate know vars x = (k, b, v, Debug x know k:e)
    where (k, b, v, e) = chk know vars x


joinMKnow = joinMaybe join
joinMVars v1 v2 = map (uncurry (joinMaybe intersect)) (
    zip v1 v2)


nullVars vars = map (const (Just [])) vars
nullRes know vars = nullResErr know vars []
nullResErr know vars err = (Just (clear know), Nothing,
    nullVars vars, err)


-- check a single item at a time
chk :: Know -> [Maybe [Location]] -> Abstract ->
    ChkState

chk know vars (Both test a b) =
       if isNothing kn1 then a2 else
       if isNothing kn2 then a1 else
       (joinMKnow k1 k2, joinMKnow b1 b2, joinMVars v1 v2
           , e1++e2)
    where
       (kn1, kn2) = iff know test

       a1@(k1, b1, v1, e1) = chkList (fromJust kn1) vars
           a

       a2@(k2, b2, v2, e2) = chkList (fromJust kn2) vars
           b


-- DEBUG
chk know vars (DoPtr assign) = (Just (clear know),
    Nothing, Just (f assign) : nullVars (tail vars), [])
    where
       f (l, Asign l2) = if (l == init l2) && (last l2 `
           elem` acyclic) then [l] else []
       acyclic = sortUnique (concat (map propAcyclic (
           sigProps (getSig know))))

chk know vars (Var field) = (Just (declare field know),
    Nothing, vars, [])

chk know vars (CallFunc _ _) = nullRes know vars

chk know vars (Test val) = (Just know, Nothing, vars, err
    )
    where err = if test val know then [Safe val] else [
       Unsafe val know]


-- the most complex case
-- try with what you know, and then try again with what you may
    know
chk know vars loop@(Loop xs) = (Just br, Nothing, vr, er)
    where
       (br, vr, er) = if optLoop == OptLoopMinimal then
           fMin know optLoopBound else fMax

       fMin know n = if isNothing cont then (fromJust
           break, vars, errs ++ [NoLoop loop Nothing])
                  else if isNothing break then (clear
                      know, nullVars vars, errs ++ [
                      DefLoop loop])
                  else if n == 0 then (clear know,
                      nullVars vars, errs ++ [FixFail
```

```
                loop])
            else if know' == know then (fromJust
                break, vars, errs ++ infVars v)
            else fMin (fromJust cont) (n-1)
    where (cont, break, v:vars, errs) = chkList
        know (Just []:vars) xs
          know' = join (fromJust cont) know

fMax = error "Todo"
```

```
infVars :: Maybe [Location] -> [Error]
infVars Nothing = error "No variables preserved,
    hence must break"
infVars (Just [] ) = [MayLoop loop]
infVars (Just (x:xs)) = [NoLoop loop (Just x)]

chk know vars x = error ("Chk: " ++ show x)
```

Appendix F:

Pasta Language Definition

This section lists the EBNF (Extended Backus-Naur form) grammar for Pasta. The issue of whitespace has been ignored. In between all expressions may be white space, or comments which start with "--" and continue until the end of the line.

The following syntax is used:

`[...]` apply zero or one times

`[...]*` apply zero or more times

`[...]+` apply one or more times

`... | ...` choose one of the alternatives

`"..."` use the literal characters enclosed

`(...)` used for grouping

```
letter ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
    "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"


fields ::= field | fields "," field
names ::= name | names "," name
terms ::= term | terms "," term
lhsExprs ::= lhsExpr | lhsExprs "," lhsExpr
rhsExprs ::= rhsExpr | rhsExprs "," rhsExpr


program ::= signature [operation]+

signature ::= name [property]* "{" [struct ";"]* "}"

struct ::= name "(" [fields] ")"

property ::= acyclic "(" names ")"

field ::= ("int" | "ptr") name

operation ::= struct block

block ::= "{" [declaration]* [command]* "}"

declaration ::= fields "=" terms ";"

command ::= lhsExprs "=" rhsExprs ";"
         | "if" "(" condition ")" command ["else" command]
```

```
         | "while" "(" condition ")" command
         | block
         | name "(" [terms] ")" ";"

lhsExpr ::= ["*"] location

rhsExpr ::= ["*"] term

prop ::= location "::" name
       | term rel term
       | "(" condition ")"

conprop ::= prop | conprop "&&" prop

condition ::= conprop | condition "||" conprop

location ::= name | location "->" name

term ::= name "(" terms ")"
       | location
       | value

rel ::= "==" | "!=" | "<" | "<=" | ">=" | ">"

name ::= letter | name letter

value ::= digit | value digit
```