Heriot-Watt University

MSc Dissertation

Animating Pointer Based Data Structures

Author:
Neil STRUTH
Msc Software Engineering,
Matriculation Number: 091605551

Supervisor:
Dr. Andrew IRELAND

August 18, 2010
0.1 Statement of Non-Plagirism

I Neil Douglas Struth confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the words of other authors in any form e.g., ideas, equations, figures, text, tables, programs etc are properly acknowledged. A list of references employed is included.

Signed:_____________________

Date:_______________________

0.2 Acknowledgements

First of all I would like to thank my supervisor Andrew Ireland for all his support throughout this project. His feedback and willingness to meet and discuss my progress was exceptional. Also Ewen Maclean for helping me get to grips with the original system and for his constant support with all matters OCaml. This dissertation would not have been nearly as complete and successful if it wasn’t for the support I received over the duration of this project. Finally I would like to thank my second reader Mike Chantler for taking the time to read and assess the dissertation.
Abstract

Understanding pointer programs is a challenge to any student of the field. Programs of this nature may seem simple on a superficial level but hide the true complexity which is found by tracing the pointers and how they interact with data. There are many ways in which one can learn about pointer programs. The most common way is to experiment with pointers by writing programs which manipulate them. This often leads to errors or unexpected outcomes due to the lack of experience and the complex nature of pointer programs. This report introduces a way of visualising pointer programs using Smallfoot, a C-like programming language which proves assertions using Separation Logic, using Animation. The report concludes that visualisation is an effective technique for reasoning about pointer programs and can reveal errors that would not be highlighted by a compiler. The main aim was to provide a tool which is able to animate many examples using a generic design which makes it extendible and applicable. This report describes the research which led to the MSc Project Animating Pointer Based Data Structures. The project introduces a way of animating Smallfoot programs which manipulate pointer-based data structures by using a formal verification technique known as assertion based program proof. The project uses Reynolds separation logic as a method of producing assertions on only the data structures affected by the code.
Contents

0.1 Statement of Non-Plagirism ........................................... 1
0.2 Acknowledgements ..................................................... 1

1 Motivations & Summary of Terms ...................................... 4
  1.1 CORE Project ....................................................... 4
  1.2 Approach .......................................................... 4
  1.3 Summary of Terms .................................................. 5
    1.3.1 Display Variables ............................................. 5
    1.3.2 Pointers ....................................................... 5
    1.3.3 Linked Lists .................................................. 6
    1.3.4 Animation .................................................... 6
  1.4 Aims ........................................................................ 7

2 Background ..................................................................... 8
  2.1 Formal Verification .................................................... 8
    2.1.1 The Literature .................................................... 8
    2.1.2 The Relevance .................................................... 11
  2.2 Static Analysis and Tools .......................................... 12
    2.2.1 SPARK .......................................................... 12
    2.2.2 ESC/Java ......................................................... 13
    2.2.3 Smallfoot ........................................................ 13
  2.3 CORE Project and Previous Work ................................ 15
  2.4 Animation ................................................................ 15
  2.5 Reflections and Literature Summary .............................. 17
  2.6 Comparison of Tools ................................................. 19
    2.6.1 Direct3D vs OpenGL .......................................... 19
    2.6.2 Programming Languages ..................................... 20

3 Requirements .................................................................. 23
  3.1 List of Requirements .................................................. 23
  3.2 Description of Requirements ....................................... 24
    3.2.1 Requirement 1 - Fully Animate all Possible Functions on Singly-Linked Lists .. 24
    3.2.2 Requirement 2 - Improve the look and feel of the system ......................... 24
    3.2.3 Requirement 3 - All Operations Must be Animated ................................. 24
    3.2.4 Requirement 4 - Be easy to use for target users .................................. 24
    3.2.5 Requirement 5 - Show the potential of visualisation software as a method of understanding programs ................................................. 24

4 Design ......................................................................... 25
  4.1 The Tool ................................................................. 25
    4.1.1 User Interface ..................................................... 25
    4.1.2 Animation ........................................................ 26
  4.2 Algorithmic Design .................................................... 26
    4.2.1 Displaying Text/Data ......................................... 26
    4.2.2 Drawing The Variables ....................................... 28
    4.2.3 Handling more than one list .................................. 28
  4.3 Structural Design ....................................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5 Summary</td>
<td>67</td>
</tr>
<tr>
<td>10 Summary</td>
<td>68</td>
</tr>
<tr>
<td>10.1 Successes</td>
<td>68</td>
</tr>
<tr>
<td>10.2 Challenges</td>
<td>68</td>
</tr>
<tr>
<td>10.3 Opportunities for Improvement</td>
<td>69</td>
</tr>
<tr>
<td>10.4 Summary</td>
<td>69</td>
</tr>
<tr>
<td>A OCaml Doc</td>
<td>70</td>
</tr>
<tr>
<td>B Top Level Interaction Diagram</td>
<td>83</td>
</tr>
<tr>
<td>C Source Code</td>
<td>84</td>
</tr>
<tr>
<td>C.1 Animation</td>
<td>84</td>
</tr>
<tr>
<td>C.2 Position</td>
<td>97</td>
</tr>
<tr>
<td>C.3 Text</td>
<td>100</td>
</tr>
<tr>
<td>C.4 Variable</td>
<td>105</td>
</tr>
<tr>
<td>C.5 Interpret</td>
<td>107</td>
</tr>
<tr>
<td>C.6 Memory</td>
<td>111</td>
</tr>
<tr>
<td>D Test Cases</td>
<td>119</td>
</tr>
<tr>
<td>D.1 Append</td>
<td>119</td>
</tr>
<tr>
<td>D.2 Copy (No Data)</td>
<td>119</td>
</tr>
<tr>
<td>D.3 Copy (Data)</td>
<td>119</td>
</tr>
<tr>
<td>D.4 Reverse</td>
<td>120</td>
</tr>
<tr>
<td>D.5 Deallocate</td>
<td>120</td>
</tr>
<tr>
<td>D.6 Traverse</td>
<td>121</td>
</tr>
<tr>
<td>E User Manual</td>
<td>122</td>
</tr>
<tr>
<td>E.1 System Requirements</td>
<td>122</td>
</tr>
<tr>
<td>E.2 Running the Application</td>
<td>122</td>
</tr>
<tr>
<td>E.3 Using the Tool</td>
<td>122</td>
</tr>
</tbody>
</table>
Chapter 1

Motivations & Summary of Terms

Software development is the process of solving problems by building applications. To do this effectively a fundamental understanding of the problem is paramount. Once the problem has been established and the software has been developed it is also important to discover whether the system built is the system which is required. To ensure this is the case, extensive testing must be done to show that the system does what it was designed to do. Testing can take many forms and can often produce unexpected results. To understand why a program has not produced the output required often involves a debugging process which takes a period of time determined by the systems complexity. It is clear that the more understanding a developer has about their code, the easier it is to debug.

Tools are often used in conjunction with traditional “dry runs” to debug algorithms. This project will aim to produce a tool capable of furthering a developers understanding of algorithms using pointer programs, a notoriously difficult style of programming to understand at “first glance”. The project uses a combination of animation and Smallfoot specifications to provide a scalable extensible tool for understanding algorithms which work on singly linked lists.

1.1 CORE Project

This project sits within an active research project the CORE Project[14]. The work done in this project builds on previous work into visualising pointer programs using Separation Logic. The main area of work is in extending this existing system to provide a more generic way of visualising pointer programs. Also animation is going to be utilised to provide a clear demonstration of how pointer programs affect data structures during execution.

1.2 Approach

The development process will be fully explained in this report. The project was most suited to an iterative approach as the functionality of the final system relied on utilising the work carried out in the beginning of the development process. Throughout the project, a number of prototypes were built which were stand alone applications which covered some but not all of the functionality required. The prototypes and their development will be presented in the design and implementation sections of this report.
1.3 Summary of Terms

This section introduces the topic *Animating Pointer Based Data Structures* and gives brief introductions to the concepts which will be used throughout this report.

1.3.1 Display Variables

This report will make reference to “Display Variables” on a number of occasions. The term is used to describe the variables which are used in the animation. The distinction between variables used in developing the system and variables which the animation displays should be made clear by the context in which they are presented. To prevent any ambiguity, the variables displayed by the animation will always be referred to as “Display Variables”.

![Display variable i](image)

Figure 1.1: Display variable $i$

1.3.2 Pointers

An essential ingredient in any linked data structure, pointers play an important role in the majority of programs. Explicit pointers are more often seen in low level programs and are a feature of languages such as C and C++. A pointer has a memory address associated with it which is dereferenced when the data value is accessed. The difference of specifying an Integer variable and a pointer to an integer is that the Integer variable will hold the actual integer whilst the pointer will hold the memory address where the data is stored.

This project is primarily concerned with singly-linked lists which are made up of multiple elements which contain a data field (which stores the data associated with the element) and a pointer which specifies the location of the next element in the list.

The pointer type is useful for representing links in a linked list but it is not without fault. Inexperienced programmers who use pointers need to be aware of dereferencing null pointers which will lead to unexpected results. This is because the program is trying to find a value which does not exist.

Pointers are a useful tool and can be used to great effect when one is writing programs which concern linked data structures.
1.3.3 Linked Lists

The project will reason about *singly-linked-lists* exclusively as they have some interesting features and utilise pointers. This section will give a brief overview of the data structure as the report will assume that the reader has some knowledge of it.

To introduce this data structure we will first look at the definition of a *list*:

A list is a finite sequence of zero or more elements.[2]

This simple data structure can be used in a variety of situations. The most common implementation is the singly linked list in which each element in a list has a pointer to the next element in the list. Common operations on this data structure include:

- Lookup
- Traverse
- Reverse
- Insert
- Delete

These operations allow for more complex algorithms for Sorting and Copying lists. These operations will be used in the remainder of the report.

![Linked List Example](image)

**Figure 1.3: An example of a linked list taken from [16]**

1.3.4 Animation

During the execution of algorithms which affect data structures, the heap (the programs memory) will go through a number of changes. The project aims to highlight these changes by producing an Animation wholly dependant of the algorithm in question. The aim is to provide a tool which will animate the changes to pointer based data structures in the heap.

This subject will be revised in later sections where the exact method used to animate will be discussed.
1.4 Aims

This section will layout the aims of the entire project:

- Provide a tool which animates changes in a programs data structures
- Take a structured approach to allow the tool to be used in most scenarios.
- Design in such a way that future work can be done with minimal changeover.

The main aims of the project are to build a tool which animates changes in programs data structures. The project will build on previous work on building snapshots of changes throughout a programs execution. The main areas of interest focus on singly-linked lists with the ultimate goal of being able to animate a copy function with data. The report will aim to show the development process of building the tool whilst highlighting the generic approach in which it was made. This approach will allow the tool to handle all possible operations on a linked list and will make any future work easier by being extensible.

The tool will need to be able to handle the following operations:

- Allocation of cells
- Deallocation of cells
- Pointer manipulation
- Multiple Lists
- Data manipulation (Being able to display and reason about data)
Chapter 2

Background

This chapter describes some of the literature which relates to this project. This chapter is split into sections which describe the main ideas of the project. This project focuses on creating a tool which automatically produces animations based on verified programs. The literature provided in this section provide the basis for the program verification which will be used in the final system. Formal verification is a large topic and there are many different approaches to it. This section describes one such a approach, assertion based program verification, and its history, to provide some context for the verification used in this project.

2.1 Formal Verification

2.1.1 The Literature

Formal Verification is an important part of this project. When trying to visualise programs we need some way of making sure these programs are correct. The literature in this section all relate to formal verification in some way. This section introduces the texts which contribute to the topic of Formal Verification.

At the end of this section the literature will be evaluated against the project to highlight their relevance.

McCarthy

McCarthy introduced a way of proving recursive functions using an assertion based proof approach in [20]. Until this paper was published, the general approach towards formal verification and proof was on proving unsolvability of functions and programs. McCarthy saw that little research had been done on the proving “algorithms that are actually used”[20]. McCarthy worked on functions and in particular recursive functions with a view that there should be one “universal programming language”. McCarthy’s work on recursive functions inspired many of the subsequent authors in the field of assertion based program verification.
Floyd

Formal verification is a powerful technique which can be used to prove that programs are correct. Using mathematics Floyd[11] laid the foundations for all subsequent works on this topic. He introduced a method of using flowcharts and propositions so that if a command which has been reached by a previous command which proposition is true, the proposition for the command reached will also hold.

“If the initial values of the program variables satisfy the relation $R_1$, the final values on completion will satisfy the relation $R_2$.” [11]

Floyd’s paper provides a way of proving programs correct using logical propositions and mathematical proofs and builds on work by McCarthy[20] who had previously done similar work on proofs for recursive functions. Floyd provided a method for verifying the correctness programs and also proving the termination of correct programs.

Hoare

In 1969, the logic provided by Floyd and McCarthy[11, 20] was expanded upon to create the logic which is used today to prove the correctness of programs. Whilst Floyd[11] applied logical formulae to flowcharts, Hoare used similar logic but applied to text[13]. Hoare introduced a notation:

$$P\{Q\}R$$

This was based on Floyd’s flowcharts[11] but expanded in to apply to more programs where making flowcharts wasn’t necessary. The notation is explained in [13]:

“If the assertion $P$ is true before initiation of a program $Q$, then the assertion $R$ will be true on its completion”

Hoare acknowledges Floyd for introducing logical Axioms in order to prove the correctness and termination of programs. In [13] he introduces more axioms which are essential to the logic which is proposed. These rules can be used to prove the correctness of computer programs. The rules are the most widely used and important tools in the field of formal verification and static analysis. They are as follows:

$$\{P[E/x]\}x := E\{P\}$$

Figure 2.1: The Axiom of Assignment

This says that all free occurrences of x in P are replaced by E. This is a fundamental axiom from which all the other logical rules are built upon.

$$\{P\}Q_1\{R_1\}, \{R_1\}Q_2\{R\} \implies \{P\}Q_1; Q_2\{R\}$$

Figure 2.2: The Rule of Composition

This rule is built upon the Axiom of Assignment. It says that if one operation holds with the precondition $P$ and post-condition $R_1$ and $R_1$ is the precondition of the second operation, then they can be combined
with the first precondition and the second post-condition.

\[
\frac{\{P \land B\} S\{P\} }{\{P\} \textbf{while } B \textbf{ do } S\{\neg B \land P\} }
\]

Figure 2.3: The Rule of Iteration

This rule allows us to prove that a repeating fragment of code is correct. However it does not say that the loop will ever terminate. This is the difference between total and partial correctness. In [11] Floyd introduces a way of proving termination by using a variant which value decreases during each iteration.

In [13] Hoare introduces a way of proving programs correct by focusing on mathematical and logical formulae. This axiomatic approach first suggested by Floyd [11] was instrumental in the rise of formal verification and static analysis.

**Burstal and Others**

The next few decades built on the work done by the authors of the 1960s. Specific areas of work include Rod Burstall’s work into introducing the Distinct predicate. Which is a precursor to the *separating conjunction* which will be used in this project. Other areas of research in this field were done which were not mentioned here. The most relevant extension to the logic introduced in the 1960s was *separation logic*

**Separation Logic**

Possibly the most significant recent development to the logic Hoare provided in[13] was suggested by by Reynolds in[24] called Separation Logic. Reynolds found that there were no reliable methods for reasoning about shared mutable data structures.

Approaches to reasoning about this technique [shared mutable data structures] have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size.[24]

The solution to this problem was introducing new logical constructs which allow us to reason about programs at a local level. This allows reasoning about the parts of the memory which the program manipulates only.

The first operation suggested by Reynolds is the Separating Conjunction:

\[ P * Q \]

That asserts that P and Q hold for *disjoint* portions of addressable storage.[24]

This is a powerful operation which allows much simpler invariants and allows for a more scalable approach to reasoning about shared mutable data structures. In this paper Reynolds also describes the *Separating Implication* which was originally suggested by Ishtiaq and O’Hearn. This operator has a lot in common with the “classical” implication operator: \( \Rightarrow \). This operation is represented by the operator: \( \rightarrow \). In [24], Reynolds describes the operator as:

\[ p_0 \rightarrow p_1 \] asserts that, if the current heap is extended with a disjoint part in which \( p_0 \) holds, then \( p_1 \) will hold in the extended heap.
The two operators \textit{separating conjunction} and \textit{separating implication} allow us to build local specifications which are scalable and applicable to shared mutable data structures.

Reynolds also describes the \textit{frame rule} which was introduced by O’Hearn in [22]. This was to counter the problem of the classical Hoare Rule of Constancy which is “not sound for separation logic”[24].

\[
\frac{\{p\}c\{q\}}{\{p \ast r\}c\{q \ast r\}} \text{ where no variable occurring free in } r \text{ is modified by } c. \text{[24]}
\]

Figure 2.4: Frame Rule

This rule is central to the project. It says that \(p\) and \(q\) will always regardless of what is true in a separate area of the heap. This rule can be used to extract the most meaningful assertions from a specification to simplify it. If one was to remove the \(r\) from the specification in the rule above we would have a specification in terms of \(p\) and \(q\) only. Using this rule will make it easier to extract the data structures from the specification without the need for having the whole specification for the entire program memory.

The operators and rules explained in this section make the program specifications more local as we can verify the parts of the heap that the code affects. Separation logic is a ideal solution to the problems with reasoning about pointers with Hoare logic. The \textit{separating conjunction} and the \textit{frame rule} allows us to extract the most relevant parts of the specification whilst maintaining the integrity and correctness of the program.

\subsection*{2.1.2 The Relevance}

This section describes how Formal Verification and the papers discussed in this section are relevant to the project.

Formal Verification is a powerful tool which can be used to prove properties of programs by using mathematics. It is the foundation for all the work in this project. The project, \textit{Animating Pointer Based Data Structures}, takes a program written in Smallfoot, a programming language used to check the separation logic specifications, and animates the changes to the data structures which the program will manipulate. Hoare and Floyd [13, 11] laid the foundations for formal verification by introducing axioms which allow us to reason about computer programs in an exact way. They improved upon an original idea by McCarthy[20] which focused on reasoning about recursive functions. Reasoning about programs allows us to accurately predict what will happen in our code. One of the aims of this project is to give a better understanding of how the users code affects memory. This is closely aligned to the goals of Floyd, Hoare and McCarthy[13, 11, 20] who succeeded in finding a method for accurately reasoning about programs which allows us to build reliable systems.

When the correctness of a program, its’ compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.[13]

Separation Logic allows reasoning about programs on a local level. This means that we can reason about the parts of the heap which the program affects, thus, greatly reducing the complexity of the propo-
sitions required for programs manipulating shared mutable data structures. Using the logic suggested by Reynolds[24], we move towards a scalable method of reasoning about programs which allows us to reason about pointer programs. The method suggested by Reynolds[24] provides an effective method for reasoning about pointer based data structures which will be used in conjunction with some analysis tools to provide the basis for the animation in this project.

Formal Verification is an important topic in the field of Computer Science and can be used to prove the reliability of computer programs. Given that it has such a strong grounding in mathematics and that five decades of research has gone into this topic alone, it will prove to be a solid foundation for the work carried out in this project.

The literature in this section only scratches the surface of this topic and contains only the most relevant pieces to this project.

2.2 Static Analysis and Tools

The project Animating Pointer Based Data Structures aims to provide an effective way for understanding pointer based programs. It is therefore important to look at some of the techniques and tools in use currently and their history. The formal methods suggested by McCarthy, Floyd and Hoare [20, 11, 13] allowed for reasoning about programs using logical proofs. Using these proofs allows us to be certain what the output of a program will be.

Static analysis is the technique of analysing a program without execution. It performs checks such as assertion based proofs, well formedness and flow control. One could say that formal verification is in itself a static analysis tool as it can be used to reason about a program without execution.

The technique of static analysis could be used in a number of ways. This section explores some of the tools and the different ways they are used. The techniques which are most widely used today aim to provide software which can be used in safety and security critical situations. This is because static analysis tools can eradicate most errors from programs and produce reliable code.

This section also describes Smallfoot, a tool which will be used in this project for automatic verification of separation logic specifications.

2.2.1 SPARK

The SPARK approach suggested by Barnes [3, 4] is a popular and robust static analysis tool which is used in the development of high integrity software systems. The ultimate goal in creating this subset of Ada was to provide a method for software development whereby any errors were eliminated at source. The strict SPARK examiner checks for errors in flow control and whether the code is well formed.

The SPARK approach also generates Verification Conditions. These are based on the Hoare axioms which were discussed in the Formal Verification section of this report. This means that by using the SPARK approach you are able to rely on the correctness of the program. This allows us to say that the program “accomplishes the intentions of the user”[13].

Using the SPARK approach allows for highly reliable and robust systems which can be used in safety-critical systems i.e., systems in which any faults or errors would result in danger or loss in life. Because it is based on sound mathematics from the field of Formal Verification this approach can be relied upon.
However, the problems highlighted in the Separation Logic section of this report still apply. SPARK
overcomes the problem of accessing shared mutable data structures by forbidding the use of pointers.
This is an effective method for maintaining the integrity of programs built using the SPARK approach.
As the SPARK examiner forbids pointers it cannot be used in the context of this project.

2.2.2 ESC/Java

Another prominent static analysis tool is Extended Static Checker for Java (ESC/Java). This tool
was introduced by Flanagan et al. [10] in 2002 as a way of statically analysing java programs using
annotations. Flanagan et al describe ESC/Java as:

> an Experimental compile-time program checker that finds common programming errors.
> The checker is powered by verification-condition generation and automatic theorem proving techniques.[10]

ESC/Java differs from SPARK as it was designed to increase programmers productivity. To do this
Flanagan et al. provide a compile-time checker, not unlike the SPARK examiner to find any common
mistakes present in the code being compiled. Such errors may be allowing dereferencing of a null
pointer. To overcome this the programmer can insert an annotation which asserts that the object being
dereferenced is not null:

```java
//@ requires input != null [10]
```

Also a novel annotation construct called an object invariant is introduced which allows the checker to
reason about objects as Java is an Object-Oriented programming language.

ESC/Java allows for static analysis of Java Programs. It shares some features with the SPARK approach
such as verification-condition generation and annotations, but the emphasis of ESC/Java is to improve
the productivity of the programmer, not to guarantee the reliability and integrity of the code.

2.2.3 Smallfoot

ESC/Java and the SPARK approach concentrated on generating and checking verification-conditions
using the axioms introduced by Floyd and Hoare [11, 13]. But these methods cannot be applied to
pointer programs as the logic is not scalable when considering shared mutable data-structures. Reynolds
suggested an extension to the Hoare logic which would be applicable in this case Separation Logic [24].
In 2005 Josh Berdine, Cristiano Calcagno and Peter W. O’Hearn introduced a tool for automatically
checking separation logic specifications, Smallfoot[5].

This tool was designed as an experiment to find out whether Separation Logic proofs could be made
automatic. The authors achieved this by using pre and post-conditions and loop invariants to prove
simple procedures using the rules and axioms of separation logic. The tool consists of an input language
which is imperative by nature. Someone using this tool could define their own procedures which would,
in some way, modify some shared resource. Provided the procedure was correctly annotated the tool
would then automatically generate verification conditions using the static analysis technique symbolic
execution. An example of the input language is provided below taken from [5]:

```
disp_tree(p) [tree(p)] /* Pre condition */ {
    local i,j;
    if (p = nil) {} 
    else{
        i:= p -> l; 
        j:= p -> r; 
        disp_tree(i); 
        disp_tree(j); 
        dispose(p);
    }
}
[emp]/* Post condition */

Figure 2.5: Function disp_tree showing the precondition and post-condition annotations.

This is a simple function which deallocates a tree from memory. It recursively disposes all left nodes of a tree, then the right and finally removes the root node. The code in the brackets [tree(p)] and [emp] refer to the pre and post-conditions respectively. It is clear from this example that Smallfoot is capable of reasoning about pointer programs. This is different from the SPARK approach which forbids the use of pointers and ESC/Java because Java has no explicit implementation of pointers. What is not clear from the disp_tree(p) example is the main aim of the Smallfoot project which was to automatically check Separation Logic specifications [5]. Berdine et al. proposed a solution to checking the specification for disp_tree(p). That was to use symbolic execution to prove the program correct:

\[
[(p -> l:x, r:y) * tree(x) * tree(y)]
\]
\[
i := p -> l; j := p -> r;
\]
\[
[(p -> l:i, r:j) * tree(i) * tree(j)]
\]
\[
disp_tree(i);
\]
\[
[(p -> l:i, r:j) * tree(j)]
\]
\[
disp_tree(j);
\]
\[
[p -> l:i, r:j]
\]
\[
dispose(p);
\]
[emp]

Figure 2.6: Proof of disp_tree from [5]

It is clear from this example that Smallfoot is able to check Separation Logic specifications. The proof above uses the separating conjunction to show that the propositions hold for disjoint portions of the heap.

Unlike SPARK, Smallfoot is not complete and could not be used for safety critical situations. Smallfoot is used primarily for checking separation logic specifications and does not have the stability of the SPARK approach. The SPARK approach is used in building security and safety critical systems and can be used for defence projects such as the Lockheed Martin C-130J.[23]

ESC/Java on the other hand provides a compile-time checker which finds and eradicates common program errors in Java programs. This is used to aid the productivity of Java developers by reducing the likelihood of errors occurring. Reducing errors is achieved by using annotations which force the user to think about how their code might be erroneous. The common theme from these three tools (SPARK Ada, Smallfoot and ESC/Java) is that they are all tools which can check and generate logical specifications. They all
achieve this using annotations which are in the form of logical assertions. Smallfoot is able to generate proofs for programs using pointers by checking separation logic specifications. This provides an effective way of illustrating the state of the data structures in the programs at intermediate stages of the programs execution.

It is also important to mention the other tools which utilise separation logic. Much work has gone into producing a number of tools which use separation logic in a static analysis context. SpaceInvader is a static analysis tool which uses separation logic to provide analysis on C programs. It works in combination with Abductor which is the tool used for generating and proving the assertions.

Other tools include SLAyer and TERMINATOR which are able to prove properties of programs which apply inductive reasoning to data structures, and prove that industrial size drivers and other software components eventually terminate. These tools are able to reason about industrial size problems in an effective and efficient manner by using separation logic. This would not be a realistic area of work if it were using the classical Hoare logic which does not account for locality. This is an area of research which is very current and could provide the basis for all subsequent work into formal methods and automated assertion based program proofs.

2.3 CORE Project and Previous Work

As stated in the introduction this project extends existing work by Ewen Maclean and others on the topic of generating and animating preconditions from Smallfoot programs built in OCaml. The existing project is situated within the CORE research project[14]. The CORE project aims to combine a variety of techniques for “automatic verification of correctness specifications”[14]. The techniques on which the project focus are: separation logic, already discussed in this report; proof planning and shape analysis a technique for analysing the structure of programs rather than their content. As this project focuses on visually analysing pointer programs using Smallfoot, which combines the techniques mentioned previously, this project fits in well with the CORE project. This is not a novel project as the main aim here is to extend the previous work within the CORE project.

The system being extended is written in OCaml and in the most abstract terms generates animations from properly annotated Smallfoot programs. This is no trivial task when reasoning about functions such as circular lists which have complex invariants which need to be generated automatically. This work was produced by Maclean who has built a system to automatically generate the logical assertions (preconditions) for iterative programs [19]. Together with Richard Addison, a MEng Industrial Placement Student, Maclean built a system which would animate a list reversal on a singly-linked list. The system in its’ current state can only deal with examples of said type. The challenge for this project is to make the system applicable to a wider range of data structures and their typical functions.

2.4 Animation

If the previous sections outlined the foundations for this project, this section describes central part. There are many ways in which we can reason about programs. The techniques for proving correctness allow us to conclude whether a program carries out the intention of the user. From static analysis, which uses mathematics and proofs to provide conclusions on the program, we move to a further level of abstraction: visualisation. When automating the process of formal verification and analysis, the goal is
to allow the user of these tools the freedom which arises from not having to perform the proofs manually. Many of the static analysis tools generate verification conditions which ultimately are used to prove some property of the program[5, 3, 4, 10]. It is not then a great leap to further abstract from the mathematical notations used in [13, 24] so that instead of trying to decipher the logic generated from these tools, the user could look upon a picture describing what is going on in their code.

Studies carried out into the use of visualisation in teaching Computer Science, agree it can aid the conceptual understanding of students.[17, 21]. The idea of visualisation as a teaching aid is not new and has been used for centuries. From the way we communicate through body-language and early mans’ cave paintings to the detailed maps, charts and graphs used today, mankind has regularly relied on visualisation as a method for communicating knowledge and information.

Take for an example Computer Science and how it is taught. One does not have to look too far to see some diagram explaining a concept in this field. As this project deals with how data is stored and manipulated using data structures, we shall explore a concept which requires a visualisation to understand fully: data structures.

Data structures are well suited to visualisation. They differ in their organisation to store and manipulate different types of data. A linked-list for example could be used to store an ordered set of elements and a simple list would store an unordered set. When we want to impart a hierarchy to our data we need a special type of data structure called a tree. Sahni defines a tree as:

\begin{quote}
\textbf{A tree} \textit{t} is a finite nonempty set of elements. One of these elements is called the \textbf{root}, and the remaining elements(if any) are partitioned into trees, which are called \textbf{subtrees} of \textit{t}. [25]
\end{quote}

It is not clear from this definition exactly what this data structure would look like. The name tree suggests a a graph expanding as from bottom to top, but actually trees are usually represented like Sahni shows:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{binary_tree.png}
\caption{Shani’s’ representation of a binary tree [25]}
\end{figure}

This is the classical diagram describing a tree. How it is organised should be much clearer after seeing a diagram illustrating an example of a tree. It is this level of comprehension that this project should aim to recreate. Illustration is not the only technique which can be used as an aid to understanding. A better technique would be animation.

Still pictures give us a snapshot of what is going at a precise moment in time. This can be useful for the purpose of supplementing an explanation with an illustration to further enhance ones point. However, the data structures which a program manipulates can change frequently during the programs’ life span. Therefore, illustrations would not be enough to show clearly what is going on in a computer program. Consider as an example a sports fan wishing to see his team at some event. If he was given one
photograph of one point in the match he would not have the same level of understanding of the game as if he had watch the match in full on the television or seen it first hand.

Animation paints a more complete picture than simple images would but it presents its own challenges. The speed of the animation is crucial to the success of the project. If the animation were just to run in real-time the user would not be able to see anything useful as the speed would be too quick (milliseconds). Also for long code with many steps it would be unreasonable to expect the user to wait for too long for a step to finish animating. These will be design decisions which will be taken once the project is running as parameters can be tweaked and experimented with.

What animation can provide is the visual sensation of movement. When a program performs functions, such as a list reversal, the pointers within that list will move from their current list item to either another item or null. To accurately reflect this at an abstract level we can use animation. Static pictures can show us the before and after situations but fail to provide us with any information as to how the before state reaches the after state. As the ultimate aim of this project is to provide a tool which lets us reason about pointer programs, we can form much stronger conclusions about what a program does by using animation instead of still pictures.

Animation by itself is not a satisfactory solution to the problem of understanding our pointer programs. Studies have shown that students learning new concepts benefit from some form of interaction [17, 21]. It is therefore important to have some level of interaction coupled together with the animation to provide the basis for understanding our pointer programs. This fits in nicely with the way the current system is implemented. Intermediate stages are animated at a line-by-line level in which the user can pause and reflect on the changes displayed by the animation. Also the source code for the programs being animated should be able to viewed in tandem with the animation to allow the user to understand what a particular line of code is doing at the level of the data structure it manipulates.

2.5 Reflections and Literature Summary

The previous sections in this chapter tell the story of the related work in the field of the project. This section summarises the literature to make a critical assessment of the topic.

Formal verification is an exceptionally large topic with firm and stable roots in mathematics. It provides us with a way of proving with certainty the output of programs. Formal verification is an important topic in the field of Software Engineering. Being able to prove properties of a program using formal methods gives rise to systems which can be relied on to perform safety and security critical tasks.

McCarthy Floyd and Hoare [20, 11, 13] all saw the importance of being able to prove that a program would successfully perform its’ operation without failure. Of course it would careless not to mention hardware and other electrical faults that could possibly occur during a programs execution which would be almost impossible to prove beyond any doubt. The methods which were introduced are still used today to produce safe systems and allow us to reason about programs.

From these formal methods which allow us to verify many programs and functions, Reynolds [24] found a solution to the problem of reasoning about shared mutable data structures. He found that the methods suggested by Hoare in [13] did not scale well to programs which were manipulating shared memory. He introduced a new logic (separation logic) to deal with this problem which scaled particularly well to problems of this nature. Pointer programs often access shared memory resources. Using separation logic we can accurately produce specifications which can verify pointer programs. As the project title
suggests, this project aims to produce a tool which can be used to gain a deeper understanding of pointer programs and formal methods by using animation. Separation logic provides a scalable method of reasoning. Having concise, scalable specifications allows us to provide automatic verification from which the animations can be produced.

Hopefully it will be clear that automatic verification provides the basis for this whole project. Applying the techniques suggested by Hoare and Reynolds [13, 24] automatically provides the basis for a deeper understanding of pointer programs through the use of animation.

From formal verification and formal methods the next logical step was to look at the tools used in applying these techniques in the field of static code analysis. This report discussed three different tools: the SPARK Approach, ESC/Java and Smallfoot. These tools all share the functionality of generating Verification Conditions which are used to prove properties of the program. Each of these tools all have different purposes. Firstly, SPARK Ada was designed to produce safe reliable code that can be used in safety and security critical situations [3]. The verification conditions generated use Hoare logic as a method of program proof. The SPARK approach has more than verification condition generation and proofs, but it is this area of the SPARK approach which is most relevant to the project.

ESC/Java was aimed at improving the productivity of the programmer by forcing the programmer to think about common errors in their code [10]. ESC/Java also generated verification conditions to prove the code correct. As the input language for this tool is Java, Flanagan et al. used a novel invariant called an object invariant to reason about objects. While this is interesting for possible further work on the project it will not be used in the current iteration.

Both of the tools mentioned above, ESC/Java and the SPARK approach, use the classical Hoare logic as a basis for program proof. It was mentioned in the formal verification section that the Hoare logic is not applicable to pointer programs as it does not scale well. Also in that section separation logic was mentioned as the solution to that problem. Smallfoot is a tool for automatically checking separation logic specifications. It does not work on safe code as it is the basis for proving pointer programs correct. This is not a problem for this project as the aim here is to produce a tool which automatically produces animations and will not be used in a safety or security critical context.

One of the most important areas of the literature review was the current system and the project which it is situated in. This project extends an existing area of work which extends Smallfoot to produce animations on a linked-list. The work by Maclean [19] and others lays the foundation for all work in this project. Having this system as starting point for the project allows the work to be more focused and should produce more striking results.

The CORE project[14] aims to provide automatic verification by combining techniques which compliment their strengths and compensate for their weaknesses. The work for this project can be placed in the CORE project as it will provide another method of automatic verification using animation.

To ensure correctness we must be able to reason about programs effectively. By reasoning about programs we gain a deeper understanding of how they work which is not unlike viewing an animation of the code. This report also mentioned the techniques of interaction and speed which are again crucial to the successful development of this project.
2.6 Comparison of Tools

This section describes the tools which will be used in the development of this project. Alternatives will be considered and presented along side the final choices for the system.

2.6.1 Direct3D vs OpenGL

Direct3D

Microsoft developed their own API for rendering 3D graphics Direct3D[27]. It was designed as a direct competitor to the open-source API OpenGL and runs exclusively on Microsoft Windows Operating Systems. Direct3D has full support for rendering primitive objects such as polygons and can be used to produce interactive animations by using Events. Animating the transparency of objects using Alpha values is also supported so the Direct3D API would be able to produce animations of sufficient complexity for the project. Also Direct3D has libraries for rendering text on the screen this is a major advantage over its competitor OpenGL which will be discussed in the next section.

However Direct3D only runs on Microsoft Windows and which is not suitable for the project. This is because the rest of the system would need to be designed to run in a Windows environment. The decision not to run the system in windows was taken in part due to the previous system which ran on a Linux Operating System.

OpenGL

The method chosen for rendering the graphics for the final system was OpenGL an open licensed API for displaying 2D and 3D graphics[28]. One of the reasons for choosing this API was that it is cross platform which means it can run on a variety of different operating systems, (Mac, Windows, Linux, Portables). Also OpenGL has a sophisticated interface for drawing 2D and 3D primitives and handles animation well. Alpha fading, anti-aliasing and other features also contributed towards the final decision as well as the fact that many languages use this API for displaying graphics.

The original system also used OpenGL to display its animation. As it contains functions for most of the features which are needed to display the level of animation required, the obvious choice was to minimise the disruption caused by replacing the current method of displaying graphics by building upon the existing system.

The main draw back for using OpenGL is that it does not have any functions for rendering text. Developers who use OpenGL often create their own fonts using display lists and bitmaps and this method will be considered further in the challenges section of this chapter.

<table>
<thead>
<tr>
<th></th>
<th>Linux</th>
<th>Transparency</th>
<th>Animation</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenGL</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Direct3D</td>
<td>×</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Figure 2.8: Table showing the capabilities of OpenGL vs Direct3D
2.6.2 Programming Languages

The main criterion for all the possible programming languages which are being considered are:

- OpenGL support
- Ease of building user interfaces (GTK+ support)
- Ease of integration with original System

Java

Java[9] is an obvious candidate for its excellent portability and graphics support. Due to its Virtual Machine a Java applet can be run in many different environments. This would be a major consideration if the final system was to be released. As it is the final system will be evaluated and used as a prototype. The main goal for the system is to be able to animate list functions. There is no plan to release the software, therefore the portability is not an issue for this project but this issue could be considered for any future work on the subject.

One of the main considerations when choosing a programming language is its support for OpenGL. Java has APIs for developing 2D and 3D graphics applets (Java 2D & Java 3D). There is also a wrapper library for developing OpenGL applications: Java Open GL (JOGL). As OpenGL was the choice of rendering technique this would provide a way of animating the final system using Java.

Interface building can be achieved with Java but it requires some thought. Having worked with Java in developing user interfaces I know first hand that creating a well structured layout using the layout managers available in Java is a challenge. One can use Swing in conjunction with OpenGL to produce the desired effect for the system but creating a user interface with this layout manager is a tedious and time consuming exercise.

The third main consideration was how easy it would be to port the current system in this programming language. The original system was designed to interpret Smallfoot programs. Smallfoot was developed using OCaml[5], the back end of the system which handles the interpretation of the input program and makes updates to the program memory was also developed in OCaml. It was due to this fact the Animation was also designed using OCaml.

Porting the animation to Java using JOGL would be a hard task indeed and would require a great deal of time. Not only would the existing animation have to be implemented in Java, the only feasible way of the program interacting with the back end would be to have two systems running concurrently: the OCaml Interpreter, Memory , etc. and the Java Animator. In doing this the animator and the back end would have to communicate frequently (after every step in the code) this would add needless complexity to the system and as such will not be used in the final system.

C++

C++[8] is another Object-oriented language which supports OpenGL. This is not as portable as Java as the system would need to be compiled on the computer or device running the system.

Building user interfaces in C++ is reliant upon the Window manager which you use. There are many available such as GLUT, Qt and SDL. The one chosen for this system is GTK+[15] which can be used to great effect to produce user interfaces. C++ supports GTK+ widget tool-kit for creating user interfaces.
As C++ can use C libraries it is very easy to use OpenGL. Unlike Java there is no "wrapper" to make OpenGL work in C++. However the problem of communicating with the back end would require significant work in order to either change the back end so that it is handled by the C++ program, or, finding a way in which the two systems could communicate effectively. This would require some sort of wrapper so that either the OCaml program could run the Animator or vice versa. As Java and C++ are object oriented it would be easier to convert the OCaml objects to C++ and Java objects. However, the original system is not object oriented in the truest sense as not all of the functions and files are contained within objects. As with Java this is overly complex and will not be used in the final system.

C#

C# ([18]) is a programming language which runs on Microsoft’s .NET framework. It has a very similar syntax to Java and like Java is Object-oriented. If the system were to be developed using C# the system would be able to run on Microsoft Windows Operating Systems. This makes this language the least portable of all the tools examined so far. Support for OpenGL again comes in the form of wrapper libraries such as OpenTK, Tao and SharpGL which allows the use of any OpenGL function. The only real benefit from using C# is the IDE, Visual Studio, is a full and sophisticated tool which can reduce the time of coding applications. The user interface builder which auto generates code is also a nice feature and makes constructing the layout of the applications windows easy. The interface builder auto-generates code for the developer which can lead to long messy and unreadable code which is an obvious drawback. For the same reasons as the previous languages and because it would only work on a Windows operating system, C# will not be used for the development if this project.

C

C is a powerful language which can be used in a variety of situations. It supports OpenGL natively and does not require any wrapper functions or libraries. User interface building is dependant on the choice of windowing tool used. This project will use the GTK+ tool so it will be as easy as building any interface using GTK+.

There are C compilers for virtually every platform so portability would not be an issue. However if the system would need to be compiled on the machine or device which it was going to run.

The most challenging aspect would be porting the original system. And whilst this task may be easier to do than the OO programming languages, due to the fact that C is imperative in nature, it would still be time consuming. It was the decision on the original project owners and myself that any time spent on porting the system would be better spent on tackling the actual problems. It was for these reasons that OCaml was the eventual choice of programming language for the system.
Objective Caml[12] is an implementation of Caml a dialect of the ML programming language. The assertion checker Smallfoot which is the input language for the tool, was developed in OCaml. OCaml has a sophisticated pattern matching system which can be effectively used as a parser.

It supports OpenGL through the library LablGl which is a direct implementation of all the OpenGL functions. It also supports building user interfaces using GTK+ through the library LablGTK. As such using OpenGL and GTK+ to display the animations should be a trivial task.

The main benefit of using OCaml/OpenGL/GTK+ for the system is that the Smallfoot and the original system were both developed in this way. This means there will be no time spent porting the original system to the new system. Of course changes and re-factoring will need to be implemented but the impact of changing the tools used will not be an issue.

Other programming languages and tools have been omitted from this comparison. This is due to the reasons for dismissing these tools are similar to the reasons already outlined in the above sections. I saw no reason to change the tools already in use from the previous project as they have all the functionality needed for this project. The only drawback is the lack of support for displaying text in OpenGL.

<table>
<thead>
<tr>
<th></th>
<th>OpenGL</th>
<th>UI Building</th>
<th>Ease of Integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>√</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>C++</td>
<td>√</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>C#</td>
<td>√</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>C</td>
<td>√</td>
<td>low</td>
<td>low</td>
</tr>
<tr>
<td>OCaml</td>
<td>√</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

Figure 2.9: Table showing the results of the comparison of the programming languages
Chapter 3

Requirements

This chapter contains the requirements for the system. As with any development project requirements play an important role. They provide direction and focus to the development process by providing clear and unambiguous details describing what the system must be able to do. These requirements will be used in the evaluation of the prototypes after they have been developed.

3.1 List of Requirements

1. Fully Animate all Possible Functions on Singly-Linked Lists.
   (a) Support Allocation or Addition of List Elements.
   (b) Support Deallocation or Removal of List Elements.
   (c) Support Multiple Lists.
   (d) Support Functions on Data.

2. Improve the look and feel of the system.
   (a) System should support Zoom Functions
   (b) Pointers should have arrows depicting what direction they are pointing
   (c) Graphics and Animation should be as Smooth as possible

3. All Operations Must be Animated.
   (a) Cells and Variables should Fade In
   (b) Cells and Variables should Fade Out
   (c) Cells and Variables should Move Position

4. Be easy to use for target users.
   (a) Interface should be as simple as possible

5. Show the potential of visualisation software as a method of understanding programs.
   (a) Animation should reflect as accurately as possible the changes in the heap during execution
3.2 Description of Requirements

3.2.1 Requirement 1 - Fully Animate all Possible Functions on Singly-Linked Lists

Requirement 1 says that all functions on a Linked List must be supported by the system. This is a key requirement and each prototype will work towards animating the functions specified in requirements 1a), 1b) 1c) and 1d). This is the main body of work for the project. Each function in requirement will be tackled in order over three prototypes.

3.2.2 Requirement 2 - Improve the look and feel of the system

At first glance Requirement 2 may seem trivial. However, it is crucial to the overall success of the system. The look and feel of the system determines how usable the system is. The Animation tool which is being developed is interactive. A system which looks good should be easy to use for the users.

Functions such as Zoom, will also make the Animator accessible to users which have different size screens or poor vision. Smooth animations will also make using the system a more pleasant experience.

3.2.3 Requirement 3 - All Operations Must be Animated

The overall goal of the project is to produce a system which can accurately animate the changes in the heap during a programs execution. Providing a snapshot of the heap at the different stages of execution is not enough for this project. The operations fade in, out and translations (changing positions) must be animated.

3.2.4 Requirement 4 - Be easy to use for target users

The need for making the system look good are almost self explanatory. Having a system which looks good however is not the same has having a easy to use interface. For that reason another key requirement of the system is that the interface should be as simple as possible to prevent the user from being distracted from the animation during use.

3.2.5 Requirement 5 - Show the potential of visualisation software as a method of understanding programs

Another key aim of the project is to show how visualisation software could be used in an educational manner. To highlight the potential for visualisation software to be deployed in this context it is paramount that the system should be accurate.
Chapter 4

Design

With any project of significant size the Design is integral to the overall success. Having an idea of what will be the most challenging aspects to implement and methods of overcoming these challenges will prove invaluable once the implementation is under way.

The key aspect of this design is to produce a reusable design which is applicable in more than one circumstance. Designing an application which applies to only one example is useful if that is the only example that will ever be used. The approach here is to design the tool in such a way as to be applicable to any example (including buggy ones) that one could imagine using linked lists.

Of course there are limits which will also be discussed. This chapter also looks at the user interface design, the animation design and the tools which will be used in building this application.

4.1 The Tool

In this section I will outline the different choices which could of been made on the animation design. The two main areas for consideration are: User Interface and Animation.

4.1.1 User Interface

The user interface in this context refers to the parts of the tool which the user will interact with directly. The most important design issues are simplicity and ease of use. The decision was taken to only have three buttons in the tool: Next Step, Zoom In and Zoom Out. The tool will have these three buttons along side the animation. Controls for moving the animation will be handled by the keyboard. The cursor movement keys move the animation in all directions. A preliminary sketch of the animation is shown in Figure 4.1.
4.1.2 Animation

The animation design is an important part as it is the interface between the user and the program's memory. The animation should be as clear and simple as possible and present the memory in an intuitive way. Most text books and papers relating to linked lists present them in a similar way. The animation will be designed in a way which is familiar to students of the field.

Figure 4.2 shows the preliminary design of the animation. It shows a three element list $i$. Element three points to NULL.

![Figure 4.2: Preliminary design of animation](image)

4.2 Algorithmic Design

This section takes some of the more challenging aspects of writing the tool and suggests ways in which they may be overcome. Strategies for tackling these problems will be presented. In cases which have more than one possible solution they will each be discussed and the best possible method will be recommended for the implementation phase.

4.2.1 Displaying Text/Data

The project will use OpenGL[28] to render graphics. In Chapter 2 it was established that OpenGL does not have functions for rendering text. The system will need to display text as it will display variables.
and where they point to. Also the heap will contain list elements which have a data value associated with them. To display the data values and variable names some functionality will need to be built into the system which will allow this text to be rendered in the final system.

OpenGL is widely used for displaying both 2D and 3D graphics in computer programs. Since displaying textual data is a common requirement for systems, either for debugging or for displaying data to the user, there are many ways to get round this problem. In this section the following methods will be considered:

- Bit-mapped Fonts
- Texture Mapped Fonts
- “Drawing” the fonts

Bit-mapped Fonts

Bit-mapped fonts “map” pixels to correspond to the character or glyph which is being displayed. Each number, letter or symbol, also known as a glyph, has its own bitmap which is an array of values which correspond to a map of pixels. They are called bit maps because a pixel is either on or off like binary data.

Bitmaps, for that reason, do not require a lot of space and are very simple to render. The only minor problem is that the fonts don’t tend to scale as well as texture mapped fonts or vector fonts.

To implement this would require making arrays for all the glyphs in the font. Then these fonts would be stored in a display list. When the program needs to draw a font, a call would be made and the appropriate display list would be called to draw the text.

Texture Mapped Fonts

Similar to bit-mapped fonts, texture mapped fonts substitute images or textures for the pixel maps used in bitmaps. To use this method a texture map which is an image containing all the glyphs in the font is loaded. To render the appropriate character, the texture map is loaded onto a quad. Getting the character desired is a case of loading the texture with the appropriate coordinates so that the desired glyph is over the quad.

According to the OpenGL website:

In many OpenGL implementations, rendering glBitmap() and glDrawPixels() primitives is inherently slower than rendering an equivalent texture mapped quad. Use texture mapped primitives to render fonts on such devices.[1]

To use this method in the final system a period of time would be required to create the texture map. Although the creation of the texture map would only be required once and could be reused or tweaked in further extensions to the system.

A draw back of this method is that variables will often “pass” each other in the process of animating. In order to stop the text from being “clipped” careful attention must be payed to the position on the z axis for each quad. Also the background colour would also have to be considered before the texture map was created to ensure that the quad would not be visible.
“Drawing” the Fonts

Another method would be to create the fonts as OpenGl primitives. This slightly less sophisticated method would require actually drawing the fonts using GL_LINE_LOOP & GL_LINE_STRIP to create the glyph required. This would also be a time consuming exercise but relatively straightforward in nature.

Since the old system used this method to display the variables, a similar approach will be continued for this project. A slight change will be to organise all the text functions into a class which has private methods for each glyph and public methods draw_string and draw_int to either draw text or numerical data. Since there is no need for symbols such as commas or periods, the text object should be quite concise.

If these symbols are required at a later stage they can be added into the class with relative ease. This emphasises the extensible nature of the design. The class could be modified to substitute the drawn characters and numbers for bitmaps if the performance or size of the data in the system is an issue.

4.2.2 Drawing The Variables

One of the more challenging aspects of the design is automatically drawing the variables. In the memory there is a store which contains the variables id and a pointer to a cell on the heap which may be null. The current setup can only handle three variable ids which are hard coded into the application i, t and o. The program checks to see whether a cell has a variable associated with it, if true then the id is checked to see if its an i t or a o. These variables are then drawn above the cell.

This method is suitable for the limited amount of examples the current system can cope with. In order to make this system more generic the animator should check the store and draw what is in it. Having the text class to draw variables will allow for any id to be drawn. However the challenging aspect of drawing the variables is associating animation parameters with the variables. Each variable will have an alpha value (for fading in and out), a current position and a previous position which will be used to animate the motion between cells.

An effective way of representing a variable would be to have a class of variables which have the alpha and position data. Then the animator would draw the variable in a position which is determined by methods in the variable object. For example if the current position for a variable is null and its previous position is a cell then fade out the variable.

4.2.3 Handling more than one list

The first few iterations will not have to think to deeply about what position the cells go because the heap will not change size. When considering operations in which cells are allocated or deallocated some decision will have to be made about where the cells are positioned in the display.

Decisions need to be taken on where to position new cells. Also if there is more than one row of cells the way the variables are positioned will need to be considered.

As described in the previous section about the animation, there will be a maximum of six cells per row. This means that if there are seven cells in the heap, the seventh cell will be placed underneath the first cell. Instead of making the Animator class make all the decisions about where a variable will be placed it would be useful to abstract this behaviour into a position list.
The position list will contain an x and y position for each cell and the cells id. So finding a position to draw the variable would be from the point of view of the animator class, a case of calling a `get_position` method on the position list, passing the id as parameter.

This will be a better way of managing the complexity of a very large heap. The zoom and infinite display area will also contribute to this.

4.3 Structural Design

Alluded to in Chapter 5 was the fact that this project builds upon an existing systems foundations. It is important to gain perspective on how this projects system sits relative to the existing system. Figure 4.3 shows the new and how it relates to the existing one. The box around the Animation, Text, Variable and Position sections shows the areas of development for this project. However, during the course of this project it is likely to be the case that some minor alterations or tweeks will need to be performed on the existing system.

![Figure 4.3: High level view of the system to be developed](image)

Figure 4.3: High level view of the system to be developed
Chapter 5

The Original System & the Input Language

The previous sections have on occasion made reference to the original system. This project aims to animate the changes in a program's memory by building upon an existing system. The structure and some functionality of the original system will be retained in the prototypes which are described in Chapters 6, 7 and 8. This chapter will describe the structure and selected functionality of the original system. Also in this chapter will be a section describing the input language which will be used as the basis for the animations.

5.1 The Original System

The original system was in place before the start of the project. Many of the structural aspects and drawing methods are retained in the final prototype. The implementation sections of the prototypes refer to selected functions which were added to the original system. This section describes the fundamentals of the system which was used as a starting point for all subsequent development in the project.

5.1.1 Structure

The system is essentially an execution engine with animation functions built on top. The system could originally support the reversal of a linked list and as such had simple “bespoke” animation functionality. The structure of the tool could be broken down into the following distinct areas:

- The Parser
- The Interpreter
- The Memory
- The Animator

The parser is, as you would expect, responsible for reading the input language (see Section 5.2) and changing it into a form which can be read by the interpreter. The interpreter then takes the statements and executes functions in the memory which is a model of the program's memory, and the animator which draws its representation of the memory which is updated after each statement has been executed. Figure
5.1 shows a diagram representing the original system. The “parsing operations” represents a number of functions used for parsing the input program.

![Diagram showing the structure of the original system.](image)

5.1.2 Drawing the Data Structure

To draw the data structure the original system would continuously draw every element in the heap. The heap is updated by the interpreter after statements which affect memory. The animator has a copy of the heap which it displays in the animation. Each element of the heap is examined to determine where to draw the pointer to the next element. If the pointer is **NULL** then the system will draw a straight line down to a black square (see Null Pointer description).
method draw_cons x y n c =
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y) ();
  GlDraw.ends();
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.ends();
  (match (fst m) with
    Cons (h,t,s) -> (if (s == 1)
      then
      (if (t == -1)
        then
          self # set_dest ((x +. 30.) , (y +. 30.)) ((x +. 30.) , (120.));
        let sit = self # get_oi () in
          self # draw_ptr (x +. 30.) (y +. 30.) ( fst (sit)) (snd(sit));
        self # move_closer ()
        )
      else
      (self # set_dest ((self # ptr_for_cons t c) ,(y +. 90.)) ((x +. 30.) , (y +. 30.));
        let sit = self # get_oi () in
          (if ((fst(sit)) < 0.) then (self # set_current (x +. 30.) (120.));
          self # draw_ptr (x +. 30.) (y +. 30.) (fst(sit)) (150.);
          self # move_closer ()
          )
          )
      )
    else
    (if (t == -1)
      then self # draw_null_ptr (x +. 30.) (y +. 30.) (x +. 30.) (y -. 30.)
      else self # draw_ptr (x +. 30.) (y +. 30.) (self # ptr_for_cons t c) (y +. 90.)
    )
    )
  )
  self # printf Printf.string "error in draw_cons - unmatched model
"构造

Figure 5.2: Method to display a list element (cons) in the animation.

The system also has to determine what side of the element the pointer will point to. If the pointer is going backwards or from right to left it will point to the right hand side of the heap element. Otherwise it will point to the left.

method ptr_for_cons n a =
  if(a < n) then
    float_of_int (((n)) * 80) + 10
  else
    float_of_int (((n)) * 80) + 70

Figure 5.3: Method to determine which side of the list element the link will point to.

Pointer animation is also handled by the system by specifying the current position and destination and incrementing the x value until it has reached its destination. The variables are drawn by examining the store to find the identifier and calling the appropriate draw function to draw the character and arrow above the element if it is pointed to.
5.2 The Input Language

It is important for anyone using the system to understand the input language. The language is procedural C-like language which is very much like the input language used by Smallfoot[5]. This section explains the syntax and the programming constructs which can be used to develop programs.

5.2.1 Syntax

This section previously mentioned the similarity between the input language for this system and the input language used in Smallfoot[5]. This guide to the syntax of the input language assumes some basic knowledge of programming and the typical constructs you would find in a C-like language.

Firstly, the input language follows the traditional convention for naming variables. A variable identifier is an alphanumeric string. In this input language variables are either pointers or integers. As such statements such as \( t = i; \) are acceptable where \( i \) is a pointer to a list element.

Since this project is only concerned with lists, this guide to the input language will focus on syntax relating to linked lists. To get the data element from a list element the syntax \( t = i \rightarrow \text{hd}; \) is used. In this example \( i \) is a pointer to a list element. The program will then assign a new variable \( t \) which will contain the data value of the list element pointed to by \( i \).

To dereference a pointer (or follow the link) a statement such as \( z = i \rightarrow \text{tl}; \) would be used. This translates as \( z \) is given the value determined by dereferencing \( i \). If \( i \) is the first element in a linked list \( z \) will point to the second element in the list.

The data and pointer values of list elements can be assigned new values using the = operator. In \( t \) is the integer value 4, then the statement \( z \rightarrow \text{hd} = t; \) would set the data value of the list element \( z \) to 4. Similarly if \( o \) is a pointer to a list element then the statement \( z \rightarrow \text{tl} = o; \) would set the pointer of \( z \) to the list element specified by \( o \).

5.2.2 Constructs

Most of the programming constructs you would expect from a C-like language are in use in the input language. Unfortunately there is one omission, function calls. At this stage the system does not support one function calling another. This is significant as it restricts the systems ability to perform the sort function. This section describes constructs using Backus-Naur Form. For simplicity a statement has not been defined.

While Loop

The input language supports while loops of the form:

\[
\text{<body> ::= \text{<statement> | <body> <statement>}}
\]
\[
\text{<while> ::= while (<condition>) <statement> | while (<condition>) \{<body>\}}
\]

An example of the while construct in use is shown in Figure 5.4.
The system also supports if conditionals of the form:

```plaintext
<ifstatement> ::= <ifthen> | <ifthenelse>
<ifthen> ::= if (<condition>) <statement> | if (<condition>) {<body>}
<ifthenelse> ::= <ifthen> else <body>
```

Figure 5.5 shows an if conditional in use.

The input language is quite restricted but can be used to animate and reason about most operations on linked lists. The omission of function calls makes it difficult to perform operations such as sort. Adding function calls to the input language would be an excellent addition for future work.

Also the initial state of the heap is hard coded into the memory. A function such as list_copy(i) (see Figure 6.2) requires a list i to be present in the heap. Another possible area for future work would be the ability to specify the initial contents of the heap using the input language. A suggestion for a program is shown in Figure 5.6.

```plaintext
1 i = llist(1->2->3->4);
2 list_copy(i);
```

Figure 5.6: Suggestion for specifying an initial heap.
Chapter 6

Prototype 1

This chapter documents the process of developing the first prototype of the project. This chapter and subsequent chapters on further prototypes are presented as self contained development cycles which contain requirements, test cases and evaluation sections specific to each prototype. The final prototype represents the system in its final state. Each prototype builds on the successes and attempts to solve any problems discovered in the previous development cycle.

6.1 Aims

In this development cycle the main focus is on how the system is presented to the user. In particular the incorporation of zoom, pointer arrows and windowing will be completed in this phase.

Along with aesthetics the prototype will build upon the existing system to incorporate allocation and deallocation. In this cycle the prototype will only attempt to develop support for static allocation and deallocation. Animation will be covered in the next prototype.

6.2 Requirements

The requirements which will be addressed at this stage are as follows:

- Requirement 1a) - Support Allocation or Addition of List Elements.
- Requirement 1b) - Support Deallocation or Removal of List Elements.
- Requirement 2 - Improve the look and feel of the system.
- Requirement 4 - Be easy to use for target users.
- Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution
6.3 Test Cases

The prototype will be evaluated against the functions included with Smallfoot. In particular the functions which perform *deallocate* and *copy* operations on linked lists will be used to test the prototype:

```c
hd,tl;
list_deallocate(i) [list(i)] {
  local t;
  while(i != NULL) [list(i)] {
    t = i;
    i = i->tl;
    dispose t;
  }
} [emp]
```

**Figure 6.1:** A function to deallocate a list from memory - Taken from Smallfoot examples [5]

```c
hd,tl;
list_copy(i) [list(i)] {
  local t;
  t = i;
  o = NULL;
  while(t != NULL) [list(o) * lseg(i,t) * list(t)] {
    z = o;
    o = new();
    o->tl = z;
    t = t->tl;
  }
} [list(i) * list(o)]
```

**Figure 6.2:** A function to copy an existing list - Taken from Smallfoot examples [5]

The traditional algorithm *copy* takes a singly linked list as input and returns a new list which is equal to the first. Equality is determined by the data which the list elements contain and the order of the list determined by the pointers or links.

This function was taken from the examples bundled with Smallfoot[5]. Smallfoot is a tool for designed to be used in shape analysis. This form of static analysis analyses the “shape” of data structures and does not reason about data. Therefore the copy function above is valid as the heap will contain two lists with equal shape.

The desired output of this function is to allocate a linked list which is the same length as the input list. At this stage the system does not support assigning data values to cells, this will be handled in subsequent iterations. The expression `o = new();` is the operation of interest which allocates a data element on the heap.

On the completion of this phase, the system should accurately depict the changes in the heap during the execution of this function. The heap will be initialised to contain a singly-linked list which has 6 elements. In the function `list_deallocate(i)` `i` points to the list that will be deallocated. It is expected that after each iteration of the loop, the rightmost element or head of the list will be removed. The loop will continue until all elements are removed leaving the empty list. The pre and post conditions also suggest that this will be the case. The precondition `[list(i)]` assumes that `i` points to a linked list. If the function terminates the heap will be empty, shown by the postcondition `[emp]`.

Windowing and aesthetic changes will be tested by simply using the functions. The changes which will be tested are:
The expected result of implementing the Zoom function is that the user should be able to zoom in and out of the display area. The changes to the design of the windowing should eliminate the causes of the error messages generated by the original system when the display window is closed. The system should also display arrows depicting the direction of the pointers between list elements.

6.4 Implementation

6.4.1 Aesthetics

After careful consideration on how best to proceed with the implementation of the first prototype it was decided that to gain familiarity with the original system and to better understand OCaml and functional programming that a relatively simple task would be attempted first. The cosmetic changes implemented in this stage make no difference to how the system calculates and displays the changes in the heap. They do however play an important role in allowing the system to be used in an effective manner.

Zoom

The system in its original configuration displayed the animation in a separate window which had a fixed size. This made viewing the details of the data structure difficult in displays with high resolutions. The ideal scenario would be to allow the user to control the size of the animation to best suit the display on which the system will be running. The best way to do this would be to implement a zoom function which would allow the user to control how much of the data structure they wish to view at any given time.

Implementing a zoom function was a straightforward task which was well suited to being the first area of development for this project. It involved a OpenGL matrix transformation using the `glScale()` function[28, 26]. A global variable `zoomfac` was created which would store the amount to zoom. So instead of actually moving the viewpoint closer to the drawing, the drawings were made bigger or smaller which would give the impression of getting closer or further away from the display. The `zoomfac` allowed for tight control over how much the user would be able to zoom in or out so that they could not zoom so far out that the animation became unreadable.

Similarly it was anticipated that at some stage the system would have to support multiple lists. It was decided that a scroll function should be implemented. Again the view was not changed for this. All the graphical elements of the system would be drawn regardless of whether they could be seen. Then a simple translation (moving the entire animation) would allow the user to choose which part of the system they would like to see.

The translation was implemented using OCaml’s OpenGL function `GLMat.translate` [12]. As with the zoom function two global variables, `lr_trans` and `ud_trans` were implemented which kept track of movement on the x (left and right) and y (up and down) axes. On each redisplay the system would then translate the entire drawn graphics to the amount specified by the two variables. The variables are currently updated via a callback method which captures a keyboard key press event although a better method would be to change the translation amount using scrollbars. The keys which “move”
the animation are the cursor keys. The up movement actually moves the display down. This is to simulate our eyes moving up over the animation. Our eyes stay in the same place and the display moves downwards. The down key is the inverse of the up key and behaves like so.

Using this method would allow for an “infinite” display area which would be able to draw many list elements. Obviously there would be an issue of readability when considering lists of too great a size. This approach further demonstrates the generic philosophy in which the system is being built. The system can cope with many lists even though it is unlikely it would ever be used to capacity.

An OpenGL program will typically have special functions called callbacks for controlling window events.

The display callback is used to render the graphics on the screen. When animating a technique called double buffering can be employed to greatly increase the efficiency of redrawing the graphics. To get the effect of animation the rate of displaying frames needs to be close to 60 frames per second [28, 26]. When frames are displayed at this rate the human brain perceives the changing of frames as a smooth animation. Double buffering speeds up the refresh rate by drawing the next frame on the front buffer whilst the back buffer is displayed and swapping after the drawing is complete. This allows the displaying and drawing of the frame to be carried out simultaneously.

The original system displayed error messages as it tried to swap the buffers after the display area had been closed. To counter this a simple if(window_open) then gl_area#swap_buffers command was added so the system could only swap the buffers if the window was open.

6.4.2 Arrows

The original animation showed the pointers as straight lines between list elements. The user had to infer which cons cell was being pointed to by looking at where the line originated. To make this more explicit, arrows were added to show the direction of the pointers. Adding arrows to the pointers in itself was a straightforward task. However, in the original configuration the pointers were animated. The pointers would move from one cell to another based on the operations specified in the program. It was important to keep the arrows pointing the correct way throughout the animation process. This was achieved by specifying the arrows position relative to the animated line. A mathematical formula was required to ensure that the arrow would change position relative to the gradient of the line. This would allow the pointer to be in any position and have a meaningful arrow describing the end point of the pointer and the direction it is pointing to.

```plaintext
(* Draws a straight line between two points with an arrow on the end. *
* Used for displaying pointers in the animation *)
method d_line xo yo xd yd =
    GlDraw.vertex ~x:(xo) ~y:(yo) ();
    GlDraw.vertex ~x:(xd) ~y:(yd) ();
    let cx = xd -. xo in
    let cy = yd -. yo in
    if(cx == 0.) then self # d_straight xo yo xd yd else
        begin
            let m = cy /. cx in
            let c = yo -. (m *. xo) in
            let chigh = c +. 10. in
            let clow = c -. 10. in
            let xhigh = (yd -. chigh) /. m in
            let ylow = (m *. xd) +. clow in
            GlDraw.vertex ~x:(xhigh) ~y:(yd) ();
            GlDraw.vertex ~x:(xd) ~y:(yd) ();
            GlDraw.vertex ~x:(xd) ~y:(ylow) ()
        end
```

Figure 6.3: Method d_line taken from animation.ml
6.5 De-allocation & Allocation

Originally the system could handle most functions on linked lists which do not change size during execution. In this stage an allocation operation would be experimented with to see what would need to be done to animate this operation. Getting the function `list_deallocate(i)` which is specified in the Smallfoot examples[5] would be the main aim of this iteration. In this stage providing snapshots of each stage of execution would be the primary aim of the prototype.

6.5.1 Static Allocation

Further investigation into the Allocation operation revealed that the current system could cope with allocations which are called by lines similar to the following. `i = new();`

The animator works by continuously drawing the current state of the heap. So, as the Interpreter and Memory included functions for adding a new list element, an allocation is possible using the system in its original configuration. What the system does not have is the ability to animate this allocation operation. As the functionality for static deallocation was already provided in the original system it will not be evaluated.

6.5.2 Static De-allocation

As stated in the previous section, the animator works by displaying the current state of the heap, therefore by modifying the contents of the heap a deallocation operation would be displayed. The system in its original configuration did not have the capacity to remove a cons cell from the heap. Achieving this was simple process which involved adding calls to functions in the Interpreter which would in turn call pre-existing functions in the Memory which would be responsible for removing the cons cell in the heap. However, allowing the dispose function which deallocates a cons cell created problems in the way the animation was being displayed. The original set up hard coded the position of the cells in the display. Therefore, pointers between list elements point to positions which don’t exist after a cell had been deallocated. Take for example disposing the first cell in a linear linked list. All the remaining cells get shifted one place to the left, so if the list elements are arranged linearly e.g. 0 points to 1, 1 points to 0 etc. a dispose operation will cause problems.

After the dispose the pointers for the cells all point to the wrong destinations. The positions of the cells remain constant because of the way the animation was originally designed. Therefore Cell 1 will point to where Cell 2 was previously not where it is now after the dispose has taken place.

A number of solutions were considered to this problem. For this prototype it was decided that to minimise the impact on the whole system, a counter would be implemented which would increase every time a dispose was performed or decrease when an allocate was performed. Then the pointers would point to one less than the original position if one cell was removed.

6.6 Evaluation

This section will evaluate the success of Prototype 1. To do this the system will be tested using the test cases from Section 6.3. Also the requirements will be revisited to ensure that the development done in
this prototype is in line with the requirements mentioned in section 6.2.

6.6.1 Aesthetics

Animation

The zoom function was implemented successfully. The prototype allows the user to control the size of the animation. This is implemented by having two buttons “Zoom In” and “Zoom Out” The following figure shows the animation before a zoom function:

![Figure 6.4: Screenshot of the animation before a zoom operation](image)

Once the **Zoom In** button is pressed, the display shows a larger version of the animation which can be moved using the cursor keys:

![Figure 6.5: Screenshot of the animation after a zoom in operation](image)

Figure 6.5 shows the effect of zooming the animation. As you can see some of the animation is not displayed. To see more of the animation the user can expand the window or move the animation by pressing the cursor keys. The size of the drawing area is doubled so that more of the animation can be displayed at the one time. We can see in Figure 6.4 that the white OpenGL drawing area is smaller than in 6.5.

Zooming out is also important in case the user is not happy with the size of the animation. What we expect to see after a zoom out is the same display in Figure 6.4 but smaller. This will give the impression of zooming out from the data structure. The results of clicking the **Zoom Out** button can be seen in Figure 6.6.
Figure 6.6 shows the effect of zooming out. The data structure remains the same as in Figure 6.4 but the graphics appear further away. This is precisely the behaviour which was intended during the development of this prototype. You can also see quite clearly in Figures 6.4, 6.5 and 6.6 the introduction of arrows to the design.

6.6.2 Static Deallocation

The expected behaviour of line 8 in Figure 6.1 (dispose t;) is that the first list element will be removed. Taking Figure 6.4 as the state of the animation before the the dispose operation, we will expect the animation to contain five elements each pointing to the next element in the list apart from the last element of the list which points to NULL. The result is shown in 6.7 below:

Figure 6.7 shows the five list elements all pointing to the next element in the list and the final element pointing to NULL. It can be deduced from this example that this prototype supports deallocation of list elements. At this stage the prototype supports removal of list elements at any position in the list. For instance the result of removing third element from the heap in Figure 6.4 would be two lists. The first list would contain elements one and two and the second list would contain the remaining elements (four, five and six). The system therefore supports the dispose t; function which removes the element pointed to by the variable t.

6.6.3 Static Allocation

Allocation is invoked by the new() function in the input language. The new() function adds a new element to the heap. Simply writing t = new() would put a new list element on the heap which is not pointed to and its own pointer points to NULL. The expected result of this operation is to create
a new element on the heap and to display it in the animation. Figure 6.8 shows the result of calling a `new()` operation in the input language. The program which is being used for the input language is `list_copy(i)` and is displayed in Figure 6.2. This function loops while there are more elements in the list and creates a new element on the heap.

The functionality which is being evaluated is the ability to display new list elements allocated on the heap. This time the evaluation starts with the initial memory configuration displayed in Figure 6.7. We see that after one new operation, the original list and a new element are displayed. The 5 elements from the original list and the new element `o` are displayed in the same row. The initial configuration for the memory was altered to evaluate the new element to make it easier to see the new element being displayed. If the configuration from 6.4 was used it would display seven elements in a row which is almost unreadable. This is an area for future work where the possibility of a position list will be considered.

It is clear from Figure 6.8 that the system supports allocation of cells to the heap.

![Figure 6.8: Screenshot of the animation after a new() operation](image)

### 6.6.4 Requirements Checklist

The requirements of a system are a useful way of determining the success of the final product. The product in question here is the first prototype which has its own specific requirements. This section reintroduces the requirements from Section 6.2 and evaluates to what extent the prototype meets them.

**Requirement 1a) - Support Allocation or Addition of List Elements**

The original system had the ability to add new list elements on the heap. As the animator draws the heap as it appears in memory the system will display the correct result of a `new()` operation.

For these reasons the system meets Requirement 1a)

**Requirement 1b) - Support Deallocation or Removal of List Elements**

The original system did not support the deallocation of list elements from the heap. This prototype added functionality for removing list elements from memory using the `dispose` statement. As the animator draws the heap as it appears in memory, this operation was also reflected in the animation.
Further evidence of the ability to deallocate elements from memory can be found in Section 6.6.2. This prototype meets Requirement 1b)

Requirement 2 - Improve the look and feel of the system

This prototype made alterations to the look in feel of the system. Section 6.6.1 describes the result of the addition of the zoom operation and the pointer arrows. Whether or not the look and feel has been improved may be subjective but the addition these new design features, in the authors opinion, improves the users experience.

Requirement 4 - Be easy to use for target users

As the target users in this case are computer programmers, the interface and the animation design should be familiar and easy to use. The interface only contains three buttons which are clearly marked.

Requirement 4 has been met.

Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution

As the animator continuously draws the contents of the memory, the animations should be accurate. This separation between the animation and the memory gives greater control over how the memory is updated.

For this prototype Requirement 5 has been met.

6.7 Recommendation for Further Work

The biggest successes of this stage was building upon the success of the the existing system to provide a more intuitive graphical display which is customisable. Allowing the user to customise the size of the animation allows for the animation to be used in a wide variety of displays. With portability being considered at this iteration it should be noted that the system is developed and tested on a Linux Operating System using the GNOME desktop environment. Further iterations should look into the possibility of using the tool in other environments.

The change to the object interaction design provides a more logical approach to displaying animations. Using this design allows the Animator to take responsibility for creating the window in which the graphics will be displayed. Further enhancements to the object oriented design could include a separate Window object which would encapsulate the behaviour and data of the window containing the animation.

An improvement could be made to the way the system handles the scroll function. Using the cursor movement keys on the keyboard is sufficient but lacks the intuitive feel of scrollbars. It is suggested here for anyone taking on this project to implement some form of scrollbar.

The solution for the pointer problem in deallocate works for the simple program list deallocate but it is not robust when considering allocating cells to the middle of the heap. To overcome that a position list could be maintained which has specifies the cons number and its position in the animation. The animator would then use this list to determine the destination of the pointer. This solution may be
implemented in further iterations. This solution would be ideal in a situation where there are multiple lists. In its current state the animator would arrange the lists from left to right in one row. This would not be easy to view. A much better solution would be to arrange the lists in a grid form. To do this the position list would contain an element for the the grid position and its position within the grid square.

The next major step would be to extend the existing system to include animation of functions which modify the size of the heap. The functions that will be addressed in the next iteration will be De-allocation and Allocation. The next iteration will focus on animating the deallocate and allocate operations using a position list.
Chapter 7

Prototype 2

7.1 Aims

One of the main goals of the project specified in Chapter 2, is to build a tool capable of animating the changes in a program's data structures during its execution. This iteration focuses on trying to build the basis for animating operations which alter the size of the data structures, in this case linked lists. The aim of this prototype is to animate the allocate and deallocate operations in the system.

7.2 Requirements

The requirements which will be addressed in the this prototype are:

- Requirement 3 - All Operations Must be Animated.
- Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution

7.3 Test Cases

The main focus is on animating the operations developed in Prototype 1 such as allocation deallocation and element movement. To evaluate this prototype Copy and Dispose will again be used. The difference in this prototype is that instead of list elements just appearing or disappearing, all the functions will be smoothly animated. This prototype does not have functionality for displaying or reasoning about data elements. As such the test cases presented here only reason about the shape of the program in a technique called shape analysis. The test cases from Section 6.3 will again be used to evaluate the behaviour of this prototype. These two operations concisely cover the functionality which is desired in animating this stage of the system. The system should animate the allocation and deallocation of list elements from the test cases provided in Section 6.3. The two examples in Section 6.3 are list_copy(i) and list_deallocate(i).
# 7.4 Animating the Variables

The original setup provided snapshots of the variables. The variables were displayed at their positions as specified by the store, (a list which specifies the variable id and the cell it is pointing to currently). At each next step event, the variable would be drawn at the position specified by the store. This setup is fine for simple applications but could become troublesome for more complex procedures with many variables. It would be more intuitive if the variables were animated as it would show more clearly where the variable was originally and where it finishes.

The goals of this stage would be for the system to fade in and out the variables instead of just appearing. Also the system will animate the movement of the variables.

To animate the changes in the variables would require getting the state of the previous and current step. This would allow the system to check the changes between the states and animate accordingly. For example, the animator will fade in the variable if its original position is `NULL` and its new position is a list element. Similarly the animator will fade out the variable if its original position is a list element and its new position is `NULL`.

The animator will translate a variable (move the variable to its new position) by drawing the variable in its original position and moving it incrementally towards its new position until it reaches its destination. As the grid system for position lists has not yet been implemented it is quite straightforward to move the variables on only the x axis.

Implementing the animation of the variables again raised the issue of the position of the list elements after a dispose or allocation has taken place. To overcome this the global variable `rem_no` which is a counter of the amount of cells which have been removed, was used to determine the position of the variables.

When the grid system is implemented the variables will have to move on two axes (x and y). This will raise further issues such as should the variables move around or through the lists which have already been drawn. Also if the drawing area is set up to include scroll bars should the animator focus on the destination position of the variable or let the user decide.

Also at this stage a new class of objects was implemented to handling the drawing of alphanumeric data (see text.ml). The reason for implementing this class was to provide a simple generic interface for the animator to draw data. The Text class has methods for drawing the characters `i`, `o`, `t` and `z`. These are called through the `draw_string` method which will display the string (provided that it is composed of the characters already mentioned) at the location specified by the objects instance variables.

```
1 method draw_string s =
2    let rec dstr ss n =
3        if (not (n = (String.length s))) then
4            try ((self # setx (x +. float_of_int (n) *. 5.)); self # draw_char (s.[n]); dstr s (n +1))
5            with CharNotSupported(c) -> ();
6        in dstr s 0
```

Figure 7.1: Method to print a string of text characters in an OpenGL drawing area - taken from text.ml

### 7.4.1 Moving toward a more generic implementation

At this stage the variables were hard coded into the system. Global variables to control the animation of all possible variables (`i`, `o`, `t` and `z`) were created. Each variable had:
• Alpha
• Translation Amount

The system up until this point would then, depending on the current and next position, draw the variables in place. This again seemed to be an instance where the system was very specific to the one problem which it could be applied. Also imagine a case where only one or two variables are used. In this case the tool would set up global variables which would never be used. There obvious problems if you look at the other extreme case. It is possible that the system would have to handle many variables, i.e. more than \( i, o, t \) and \( z \). Say the program was extended to support the entire alphabet as variable names. For each letter in the alphabet the system would have at least three variables storing information on transparency and the amount the variable has to move to reach its final position. Clearly this is not the most elegant or efficient solution for animating variables.

The main challenge for making the operation of drawing variables more generic is that each variable used in the program will need system variables to control the transparency and the translation amount. The chosen solution was to create a simple variable class which would encapsulate these values and determine the position on the screen where the variables will be drawn. The full listing of the variable class can be found in the Appendix.

Whilst the system is running a list of variables is maintained. The list is set up every time the animator is updated. This functionality is handled by the method \texttt{setup.vars} shown in Figure 7.2.

```ml
1 method setup_vars () =
2 let nvars = self # get_vars next_store in
3   Printf.printf "nvars size = %d\n" (List.length nvars);
4   let rec sp lst =
5     match lst with
6     | [] -> []
7     | (hd::tl) -> new variable hd::sp tl
8     in vars <- sp nvars;
9
10 let rec set_npos lst =
11    match lst with
12    | [] -> ()
13    | (hd::tl) ->
14      hd # set_npos (cur_pos # get_cons_pos (self # get_var_pos (hd # get_id) next_store));
15    set_npos tl
16    in set_npos vars;
17
18 let rec set_o pos lst =
19    match lst with
20    | [] -> ()
21    | (hd::tl) ->
22      hd # set_o pos (cur_pos # get_cons_pos (self # get_var_pos (hd # get_id) cur_store));
23    set_o pos tl
24    in set_o pos vars;
```

Figure 7.2: Method \texttt{setup.vars} () from animation.ml, initialises the variables objects as they are needed

The global variable \texttt{vars} is a list of variable objects. For each variable in the store, a new variable object is created. The two functions \texttt{set_npos} and \texttt{set_o pos} set up the new position and original position respectively. This is a much more elegant solution. By creating the variables on the fly the system only has to make space for the variables which are in use at any given time.

To draw the variables a new text object is created which has the position of the current variable object being drawn. The method \texttt{draw_vars} (Figure 7.3) handles the drawing of the programs variables:
method private draw_vars =
  let rec draw lst =
  let t = new text 0. 0. in
  match lst with
  | [] -> ()
  | (hd::tl) -> (
    if(hd# check_fadein ()) then
      (hd# reset_alpha (); hd# inc_alpha ());
    if(hd# check_fadeout ()) then
      (hd# dec_alpha ());
    if(hd# check_move_right ()) then
      (hd# inc_trans ());
    t# set_alpha hd# get_alpha;
    t# setp hd# get_dpos;
    t# draw_string hd# get_id;
    self # draw_v_arr t# getx t# gety; draw tl)
  in draw vars

Figure 7.3: draw_vars method from animation.ml, draws the display variables

This method calls the recursive function draw on the list of variables which is set up after each update. The function checks if the variable needs to be faded or animated and increments the appropriate member variable of the variable class. Then the text object position is set to the position of the variable and the variable id along with the arrow which points to the list element is drawn.

7.5 Animating the Elements

To make the application more generic the focus switched from animating lists which have a maximum size to lists which can grow.

7.5.1 Animating the List

The animation of the data structure followed a similar pattern to the animation of the variables. All the animation operations such as fade in, fade out and move were again implemented for the cells in the list. Again two lists were updated throughout the execution of the program: current heap and next heap. The cells were then faded or moved based on the comparison of these two lists. The main hurdle to overcome in this stage was how to shift all the cells to the left of the current cell being disposed. This was achieved by altering the method draw_data_struct so that instead of drawing the entire heap it would draw a segment of the heap. The new method draw_data_struct_seg takes two integer parameters specifying the start and end positions. Based on these positions the animator then draws the data structure in the correct position of the display.

Being able to draw segments allows for greater flexibility when animating the list.

7.5.2 Dispose

A dispose operation has the following steps in the system:

- Draw data structure segment until affected list element
- Draw affected element
- Draw remaining data structure
- Fade out affected element
OpenGL[28] works by continuously redrawing shapes onto the display. Therefore to animate successfully you have to specify the colour and the amount to move every time it is drawn. The draw_data_struct_seg method allows us to specify separate colour (transparency) and translate(move) values for each segment of the data structure.

7.5.3 Allocation

The complexity of the animation is increased significantly when considering allocation. The problem is where do we put a cell after it has been allocated? The previous configuration would place the newly allocated cell to the right of the last cell drawn. Imagine a scenario where we start with a list of 10 elements all side by side. If we were to perform the copy operation on that list we would end up with two lists with 10 elements each all on the same row. It should be clear that having too many elements in one row makes it difficult to view the details of what is happening to the memory.

To overcome this problem one of the design suggestions was implemented to deal with this added complexity, the Position List. This simple concept of maintaining a list of all the positions of the cells drawn allows the application to make decisions on where cells should be placed. The simple rules for determining the positions of cell is as follows:

\[
y \leftarrow \text{amountOfRows} \\
x \leftarrow \text{columnsAtY} \\
\text{if } x \geq 5 \text{ then} \\
\quad y \leftarrow y + 1 \\
\quad x \leftarrow 0 \\
\text{else} \\
\quad x \leftarrow x + 1 \\
\text{end if}
\]

The position list first finds the maximum y position, the last row. If the the last row has 6 elements then the new cell will be the first position on a new row placed underneath the last row. Other functionality must be implemented in order to maintain this position list. The functions add, remove and get are also implemented in this stage.

Instead of making the Animator maintain the position list, it would be better design to create a class position list to encapsulate the data and behaviour required to initialise and update the positions as necessary. It was for these reasons that a Position class was created which has the following instance variables:

- lst
- hp

lst is the list of all the list elements and their positions. This list is updated when elements are added or removed from the heap. All entries in the list are of the format (id, x_position, y_position). A list element with id 4 placed on the first column of the second row would have (4, 0, 2) as its entry on the position list.

When a position object is created it takes a list as a parameter:

class position_lst (heap:(anim_datastructure) list)
The parameter heap represents a copy of the contents of the memory at the moment the object is initialised. This is used to initialise the position list. The animator has a copy of the heap from the memory object which is updated after every step in the animation. The animator sets up the position list once when it is initialised by passing the model of the memory it receives at the start of the program. Once the list has been initialised the hp variable is not used as all updates to the position list are handled by the member variables of the position class. The animator after every operation which changes the position of the heap cells will call the appropriate function on its position object.

The algorithm for used by the animator to allocate a cell is as follows:

1. Draw data structure up to but not including the element being allocated
2. Set alpha to 0 (Completely transparent)
3. Draw element being allocated
4. Draw the variables
5. Set flag to increment the alpha value over time

This is shown in the private method `allocate_cons` shown in Figure 7.4

```plaintext
(* Method to animate the allocation of a list element in memory *
@param id the integer identifier of the list element *)
method allocate_cons id =
  unfinished_al <- true;
  apos <- id;
  (* draw the data structure up to the position we wish to allocate *)
  self # draw_data_struct_seg 0 (id - 1);
  (* set color to completely transparent and draw element *)
  GlDraw.color ~ alpha:alphas.(1) (0.0, 0.0, 0.0);
  self # draw_data_struct_seg (id) (id);
  (* if element completely opaque then animation has finished *)
  if alphas.(1) >= 1. then (allocate <- false;);
  GlMat.pop;
  self # draw_vars
```

Figure 7.4: Method `allocate_cons id` taken from animation.ml

This method is only visible to the object so it is labelled a private method. Specifying the access level ensures that no other object can call this method. This is important because the process of animating an allocation operation is only performed by the animator. The flag `unfinished_al` is set to allow the user to skip the animation. If the animation is not skipped, once the cell is fully opaque the method `finished` is called which adds the cell to the position list:

```plaintext
(* Method to animate allocation of a list element in memory *)
method finished pos disp =
  if disp then
    begin
      Printf.printf "Removing %d\n" pos;
      cur_pos#remove_pos pos;
      self#setup_vars ();
      animate_cons <- false;
      unfinished_dp <- false;
    end
  else
    begin
      Printf.printf "Added %d\n" pos;
      cur_pos#add pos;
      self#setup_vars ();
      allocate <- false;
      unfinished_al <- false;
    end
```

This method is only visible to the object so it is labelled a private method. Specifying the access level ensures that no other object can call this method. This is important because the process of animating an allocation operation is only performed by the animator. The flag `unfinished_al` is set to allow the user to skip the animation. If the animation is not skipped, once the cell is fully opaque the method `finished` is called which adds the cell to the position list.
This method is called whenever a dispose or allocate operation is performed. The method takes two parameters: \texttt{disp} is a boolean value which is used to determine if an allocate or dispose operation is being performed. The appropriate cell is then added to the position list at the next available position.

### 7.5.4 Animating the intended variables

Animating the data structure using the methods shown in Sections 7.5.2 and 7.5.3 creates problems when considering the display variables. Consider the case where there are display variables before and after the list element which is being disposed. The data structure is drawn up to the point of the element which is being removed. Every element after this point is shifted one place to the left. The line \texttt{self\#draw_vars} is called within the animation loop, therefore all the variables in the animation are shifted one place to the left.

To overcome this the variables which are not to be shifted should be drawn before the animation loop and the variables which are on the same list and appear after the element being removed are drawn in the animation loop. This was implemented using two methods which are shown in Figure 7.5 and 7.6.

```
method draw_static_vars pos =
let rec draw lst =
    let t = new text 0. 0. in
    match lst with
    | [] -> ()
    | (hd::tl) -> if (snd(pos) = snd(hd#get_npos)) && (fst(hd#get_npos) >= fst(pos)) then
draw tl else
    if(hd#check_fadein ()) then
        (hd#reset_alpha (); hd#inc_alpha ());
    if(hd#check_fadeout ()) then
        (hd#dec_alpha ());
    if(hd#check_move_right ()) then
        (hd#inc_trans ());
    t#set_alpha hd#get_alpha;
    t#setp hd#get_dpos;
    t#draw_string hd#get_id;
    self#draw_v_arr t#getx t#gety; draw tl))
in draw vars
```

**Figure 7.5:** Method to display the variables which are not animated - taken from animation.ml

```
method draw_mov_vars pos =
let rec draw lst =
    let t = new text 0. 0. in
    match lst with
    | [] -> ()
    | (hd::tl) -> if (snd(pos) = snd(hd#get_pos)) && fst(hd#get_pos) >= fst(pos) then(
draw (hd::tl))
    if(hd#check_fadein ()) then
        (hd#reset_alpha (); hd#inc_alpha ());
    if(hd#check_fadeout ()) then
        (hd#dec_alpha ());
    if(hd#check_move_right ()) then
        (hd#inc_trans ());
    t#set_alpha hd#get_alpha;
    t#setp hd#get_dpos;
    t#draw_string hd#get_id;
    self#draw_v_arr t#getx t#gety; draw tl) else draw tl))
in draw vars
```

**Figure 7.6:** Method to display the variables which need to be animated - taken from animation.ml

The method \texttt{draw_static_vars} checks to see if the variables appears either in another row or before the affected list element. If that is the case the variable is drawn. \texttt{draw_mov_vars} is the opposite and draws the variables which will be moved by the animation.
7.6 Evaluation

At the end of this development cycle the system should fully support all functions on linked lists which do not manipulate data. This section will evaluate the performance of the system on two functions which test the systems capability which were introduced in Section 7.3: list_copy(i) and list_deallocate(i). Unlike the previous prototype these functions will be run from start to finish to fully test the expected behaviour of the system. The success of this prototype will be determined by the accuracy of the system and the ability to animate the changes in the programs data structures.

7.6.1 Copy List

In this section the function copy_list (see Figure 6.2) will be run to conclusion. The function takes an initial linked list i and produces a new linked list o which is the same length as i.

The first line of list_copy(i) assigns a new variable which is equal to i the pointer to the first element of the linked list:

```
(a) Animation
Heap (Step 1) :
{  
  0 -> (Data:-1) 1  
  1 -> (Data:-1) Null 
}

(b) Heap
```

Figure 7.7: Screenshot of the system after the statement t = i;

The next step is for the program to create a new NULL variable o. This will serve as the pointer to the new list the program is creating. The next line is the start of the while loop. As t is not equal to NULL the loop is entered and a new variable z is created which is equal to o (NULL). The variable o is then assigned to point at a new list element. The result of the o = new(); is shown below:

```
(a) Animation
Heap (Step 2) :
{  
  0 -> (Data:-1) 1  
  1 -> (Data:-1) Null  
  2 -> (Data:-1) Null 
}

(b) Heap
```

Figure 7.8: Screenshot of the system after the statement o = new();

The next line assigns o’s pointer to location of variable z. Since z is NULL no change will be made to either the memory or the animation. The variable t is moved to the list element which it points to in the line t = t->tl;. This result is shown in Figure 7.9.
As \( t \) does not point to NULL the loop continues. The variable \( z \) is then moved to the list element pointed to by \( o \) in the line \( z = o; \)

A new list element \( o \) is then created. This is shown below in Figure 7.11

You can see from the last screenshot figure 7.11 that the program has successfully created two new elements. The next set of instructions changes the pointer values. The pointer for the last element created changes from NULL to the element pointed to by \( z \) in the line \( o \rightarrow tl = z; \). The variable \( t \) is moved to point to the element pointed to by \( t \). Since this is NULL, \( t \) becomes NULL and the program terminates leaving two lists of equal size \( i \) and \( o \).
Figure 7.12: Screenshot of the system after the statement $o \rightarrow tl = z$;

Figure 7.13: Screenshot of the system after the statement $t = t->tl$;

7.6.2 Deallocate List

This section walks through the function `list_deallocate(i)`. Again there is a two element list $i$ initially in the heap. The function removes a linked list from memory. Instead of showing every step in this function this evaluation will look at the ability to remove elements from memory. As such steps which only manipulate variables and do not remove or move list elements will be discussed but not shown.

The function is basically a while loop and will iterate until the list is `NULL`. The line `while(i!=NULL){` shows this. The first statement is to make a new variable $t$ which points to the element $i$ points to. The next step is to move the variable $i$ to the element to which it points. After that is complete the element pointed to by $t$ is removed from memory by the line `dispose t;`. 

Figure 7.14: Initial heap in `list_deallocate(i)`
The next step is to assign the value of \( t \) to be \( i \) (a pointer to the next list element). The variable \( i \) is then assigned to the value which it points to (NULL). The program then removes \( t \) from the heap with the instruction `dispose t`; leaving the empty heap.

The program then terminates as \( i \) is now NULL.

### 7.6.3 Requirements Checklist

This section revisits the requirements from Section 7.2 to show that Prototype 2 addresses the issues specific to this stage.

**Requirement 3 - All Operations Must be Animated**

An important requirement for this prototype, Requirement 3 states that all operations should be animated. In the sections 7.6.1 and 7.6.2 the operations such as the fading in and out of list elements and variables and the moving of list elements and variables were all tested successfully. Also pointer movement which was already implemented is animated.

For these reasons Requirement 3 has been met.
Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution

A change to the way the data structures are drawn means that testing was necessary to ensure that the entire heap was drawn at every stage of execution. Again sections 7.6.1 and 7.6.2 show the heap alongside screenshots of the animation in progress.

As the animation in these sections reflect the heap, Requirement 5 has been met.

7.7 Recommendation for Future Work

Extending the variable class is simply a case of adding functions for the other letters in the alphabet i.e. draw_a, draw_b etc. Provided the class still has the functions draw_string and draw_int the text class could be extended to use bitmaps or texture maps.
Chapter 8

Prototype 3 - Final System

8.1 Aims

One of the main requirements of the system is the ability to reason about data. In a linked list each element comprises of:

- A pointer to the next element
- Data

When performing operations such as a sort or a copy we are really interested in the data. The previous prototypes focused on operations on such as copying the shape of a linked list. This prototype aims to improve on the system by adding functionality to the interpreter and animator to be able to support copying the contents of a linked list.

8.2 Requirements

The requirements met for this final prototype will encompass all the requirements listed in Chapter 3. However, the requirements which will be given special consideration in this section are as follows:

- Requirement 1d) - Support Functions on Data
- Requirement 3 - All operations should be animated
- Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution

8.3 Test Cases

The example which would be used as a target for the prototype which will developed for this iteration is a modified version of the copy which makes an exact copy of the list including the links and data elements. The algorithm is as follows:
8.4 Data

8.4.1 Animator

The main challenge in this iteration is to develop a method of displaying data. For the reasons discussed in the design a class was developed which handles the drawing of data (see text.ml in the appendix). The public methods of the text class would allow the animator to draw data using a simple interface.

As mentioned in Section 7.4 a class was developed to support the drawing of data in an OpenGL display area. To support numeric data the class was modified to include methods for drawing the digits 0-9. This would allow the object to draw as many digits as required using the method `draw` (see Figure 8.2). Although it is possible to draw a great range of integer values a decision was taken to make the system support the drawing of up to two digit integer values (i.e. 0-99). Any more than two digits makes the system cluttered and virtually unreadable.

```plaintext
method draw d =
  if d >= 0 then
    ignore(*
      if d > 9 then
        begin
          let tens = d / 10 in
          let units = d mod 10 in
          self#draw_digit tens;
          self#draw_digit units;
        end
        else self#draw_digit d
      end
    )
  end
end
```

Figure 8.2: Method to print integers in an OpenGL drawing area - taken from text.ml

The method `draw` calls a private method `draw_digit` which is responsible for drawing the individual
digits of the integer.

```ocaml
digit d =
match d with
| 0 -> self#draw_zero
| 1 -> self#draw_one
| 2 -> self#draw_two
| 3 -> self#draw_three
| 4 -> self#draw_four
| 5 -> self#draw_five
| 6 -> self#draw_six
| 7 -> self#draw_seven
| 8 -> self#draw_eight
| 9 -> self#draw_nine
```

Figure 8.3: Method to print the individual digits in an OpenGL drawing area, called by the method `draw` - taken from text.ml

To display the data the animator object simply creates an instance of the text class with the desired position as arguments to the class constructor and calls the `draw_int` method like so:

```ocaml
match h with
| Data (d) -> let text = new text (x +. 25.) (y +. 80.)
in text#draw d;
```

8.4.2 Memory/Interpreter

Drawing the data values of list elements was made a relatively simple task with the introduction of the text object. However getting the system to parse and interpret programs which reason about data proved to be more problematic. The challenge for this prototype was parsing the copy program listed in Figure 8.1.

To locate the problems with the original system, the input program in Figure 8.1 was used to see how far through the program it would animate. Line 6 in the program, `if (c==NULL) {z=NULL;}` was the first stumbling block. Further investigation into the interpreter showed that if else conditionals were not supported by the interpreter. Fixing this required catching the pattern for an if statement in the `next_step` method in the interpreter and adding the appropriate statement to the current statement list.

```ocaml
(* process the next statement *)
method next_stmt =
match cur_stmt with
(* -- Other input functions omitted -- *)
(Pstm_if (exp ,stm1 ,stm2)) ->
(Printf.printf "If: \n";
(match (self#eval_exp exp) with
(Bool (x)) -> if (x == 1) then (Printf.printf "condition met...\\n"; cur_stat <- [stm1] @ tl)
else (Printf.printf "condition not met...\\n"; cur_stat <- [stm2] @ tl)))
(*--- More input functions omitted --*)
```

Figure 8.4: An extract from the `next_step` function in interpret.ml

When an if statement is encountered the conditional statement is evaluated. If the statement evaluates to `true` the true block is appended to the statement list. If, however, the statement evaluates to false the false block is appended to the statement list.

The next problem was that the interpreter did not have the ability to assign data to list elements. If the heap was set up correctly, the list could contain and display data. In our example the program assigns
values from one linked list into new list elements which eventually make up the new copied list.

Lines such as \( t = i->hd \); which creates a new integer element \( t \) with the data value from \( i \) would not be supported in the current configuration. To build support for statements such as these a further modification to the \texttt{next\_step} method was required.

Previously only pointer values could be the right hand side of an assignment statement. As with most of \texttt{next\_steps} statement processing the assign statement was processed using OCaml's pattern matching function.[7] If the value on the right hand side of the assignment statement is not a pointer value then an \texttt{integer} value is assigned to the variable on the left hand side. This is done by calling the appropriate functions in the memory object:

```ocaml
(* process the next statement *)
method next_stmt =
match cur_stmt with
  (* -- Other input functions omitted -- *)
  | (Pstm_fldassign (exp1, comp, exp2)) ->
    (Printf.printf "Field Assignment:\n";
      (match mem # get_field_type comp with
        PTR -> (self # get_var exp1);(mem # fldassign (self # get_var exp1) comp (self # get_var exp2))
        _ -> (match (self # eval_exp exp2) with
          Int(x) -> (mem # nfldassign (self # get_var exp1) comp x)));
    (anim # update "_nil_";cur_stmt <- tl);
    (--- More input functions omitted --*)
```

Figure 8.5: A further extract from the \texttt{next\_step} function in interpret.ml showing the data assignment matching

More work was done to the memory object to fix bugs which had not been tested due to this functionality being new to the system. Incorporating the data into the memory appeared to have been attempted but not fully implemented. This was to be expected as the original system was used as solid starting point for all development in this project.

8.5 Evaluation

8.5.1 Copy List

In this section the test case \texttt{copy\_list(i)} which is listed in Figure 8.1 will be run. The heap will initially be configured to have three elements which point to the next numbered element: Element 0 will point to Element 1 and Element 1 will point to Element 2. Each element in the heap initially has the value of its position in the list: The first element has a data value of 1 and the second element has the data value 2.

The separation logic specification for this program shows that if the heap initially contains a list \([\texttt{list(i)}]\) then if it executes successfully the heap will have two lists \( i \) and \( z \): \([\texttt{list(i)} * \texttt{list(z)}]\).

The aspects of the system which will be evaluated are whether the system can effectively display and manipulate data values. Also the accuracy of heap will also be tested. We expect to see the second list \( z \) to not have sequential id numbers for its list elements. This is because space has to made in the heap for the temporary variable \( t \) which is used to store the data values from list \( i \).

Figure 8.6 shows the heap in its initial configuration. The two list elements are shown each with their data element and pointer. We can see from Figure 8.6 that the system already can display data successfully. In order to determine whether the system can support the operations required by the function \texttt{copy\_list} the program needs to be run further.
The next step allocates a new element onto the heap. This new element \( z \) should have a \textbf{NULL} data element and a \textbf{NULL} pointer or link. Figure 8.7 shows the result of this step:

\[
\text{Heap (Step 1):} \\
\{ \\
\quad 0 \rightarrow \text{(Data:1)} \ 1 \\
\quad 1 \rightarrow \text{(Data:2)} \ 	ext{Null} \\
\quad 2 \rightarrow \text{(Data:-1)} \ 	ext{Null} \\
\}
\]

Figure 8.7: State of heap in \textit{copy_list(i)} after 1 step

The next line: \( z->\text{tl}=\text{NULL} \); sets the pointer element of \( z \) to \textbf{NULL}. As this is already the case, no change to the heap or the animator is performed in this step. The program then creates an integer in memory which is a copy of the value of \( i \) in the line \( t = i -> \text{hd}; \). The animator should now fade in a new element which \( t \) which has the data value of the first element in \( i \). The pointer of \( t \) will be \textbf{NULL}. The result of this step is shown in Figure 8.8:

\[
\text{Heap (Step 2):} \\
\{ \\
\quad 0 \rightarrow \text{(Data:1)} \ 1 \\
\quad 1 \rightarrow \text{(Data:2)} \ 	ext{Null} \\
\quad 2 \rightarrow \text{(Data:-1)} \ 	ext{Null} \\
\quad 3 \rightarrow \text{(Data:1)} \ \text{Null} \\
\}
\]

Figure 8.8: State of heap in \textit{copy_list(i)} \( t = i -> \text{hd}; \)

Then the program sets the value of \( z \) to the value of \( t \) in the line \( z->\text{hd} = t; \). The result is shown in 8.9.
In the next two lines of the program \( o = z; \ t t = i -> t l; \) the program assigns two new reference variables \( o \) and \( t t \) which point at the address of \( z \) and the dereferenced location of \( i \) (where \( i \) points to).

The next step in the program is a while loop while (\( t t != \text{NULL} \)). The variable \( t t \) is a pointer to the pointer value of \( i \). This loop says while \( i \) does not point to \text{NULL} then perform the loop block.

Since \( i \) is not a singleton list the loop is entered. The first step in the loop is to create a new element on the heap through the statement \( o o = \text{new}(); \). After this statement the expected output is to have five list elements: the elements from the previous step plus the new element \( o o \). The result is shown in Figure 8.10:

The output from the animator is as expected. The next line \( o o -> t l = \text{NULL}; \) makes sure that \( o o \) does not point to any other element. This will always be the case because when a new element is created, it has \text{NULL} for both its data and pointer element. The next line is more interesting however, as it changes the value of the local variable \( t \). The line \( t = t t -> h d; \) sets the value of \( t \) to be the value of the variable \( t t \). The way this is done is by first removing the old integer \( t \) from the heap and creating a new integer \( t \) which has the value of \( t t \).

The results of this operation are shown in 8.11:
Figure 8.11: State of heap in `copy_list(i)` after `t=tt->hd`;

The textual representation of the heap shows that the original value of `t` was removed from memory. We can see this because the old integer `t` had the id 3 which no longer exists on the heap. The new integer `t` has the id 5 and the value 2.

In the next step, `oo->hd=t`; the element `oo` is given the integer value of `t`. The next step involves making sure `o` points to the correct element. `o->tl=oo` ensures that this is correct as `oo` represents the next element in the list.

Finally `o` moves to the next element in the list `o=oo`; and `tt` moves to the next element in the original list. Since there are no more elements in `i` the program terminates leaving the original list, a copy and an integer in memory.

The final result is shown in Figure 8.12

This program was deliberately chosen to be the basis for evaluating this prototype due the amount of animation functions which are needed to run it. The program requires allocation of heap elements, deallocation of elements, displaying data, changing data values and variable animation. Given the diversity of operations required it seemed the perfect choice for the final prototype of this project.

As with the previous prototypes before it, this prototype essentially has a two-part structure: the animation and the memory. Testing and evaluating the performance of the animation is a particularly challenging task in a paper based report such as this. It is difficult to show on paper the way the system moves certain elements of the display. For that reason this evaluation section merely shows the system after the animation operation has complete in intermediate steps.

From the many discussions that have taken place over the final system it is suffice to say that the animation section has been generally well received.
8.5.2 Requirements Checklist

This section checks that the prototype meets the requirements listed in section 8.2.

Requirement 1d) - Support Functions on Data

Perhaps the most important requirement of Prototype 3 is the ability to perform operations on data. Section 8.5.1 shows the function `copy_list` in action. As we can see from the screenshots the system fully supports data operation on linked lists.

For that reason Requirement 1d) has been met.

Requirement 3 - All operations should be animated

Requirement 3 was met in Prototype 2 (see Section 7.6.3. This prototype is concerned with accurately displaying and changing data values in a linked list. It was decided that the change in data would not be animated as too many animated operations could slow down the animation as a whole.

As animating data was not one of the requirements listed in Chapter 3 Requirement 3 has been met.

Requirement 5 - Animation should reflect as accurately as possible the changes in the heap during execution

The only change in this prototype to previous prototype was the ability to display data. Section 8.5.1 shows the data being displayed in the animation alongside a textual representation of the heap. As the data values in the heap are being displayed in the animation Requirement 5 has been met.

8.6 Recommendations for Future Work

As this is the final prototype in this project there will be no recommendations for a further prototype. Chapter ?? outlines some suggestions for anyone wishing to carry on the work done in this project.
Chapter 9

Future Work

Many development projects start off by using previous research in order to accomplish their goals. This project built on top of previous work in the field (see Chapter 5). In this chapter some possible areas for work to improve upon this system will be suggested for anyone wishing to carry on this area of research. Suggestions for change in the way the system is animated will be the main focus for future work.

9.1 Trees

One of the major constraints of the system is that it can only handle operations on singly-linked lists. A useful extension to the system would be to add support for other data structures. The system works on linked lists and unordered lists. Possible extensions would be to first implement support for doubly linked lists and to extend that further to support trees.

Implementing doubly-linked lists would be a case of adding a new type into the memory which has two outgoing pointers and a data element. The pointers would represent the next and previous element in a list. The animation would also need to be changed to have list elements with two pointers instead of one. Currently the list items appear as two “boxes” one for data and one for the pointer. The doubly linked list would require three “boxes” for each list item (pointer, previous, next).

Binary Trees can be thought of as special doubly linked lists as they have a left and right node. The main difference between lists and trees is that trees store hierarchical data, i.e., there is a parent-child relationship between data elements. As such the memory would have to be altered to contain tree elements. Also the animation would have to modified to display a tree. Operations such as deallocate, allocate would have to be modified to allow for movement on the y-axis, i.e., up and down.

However most of the work in this project was done in the most generic way possible, therefore, the Position List could be used to display information about trees. Since the position list stores information on the vertical and horizontal position of elements, no real change would need to be here. You could imagine a complete tree with three elements having the position list = \[(0,0), (0,1), (1,1)\] This may need to be tweaked so that the when we are dealing with trees the finds the position of the parent and displays the left child one place to the left and the right child one place to the right: = \[(0,0), (-1,1), (1,1)\]. Similarly all the variable and text functions could be applied to a examples using trees.
9.2 Lists of Arbitrary Length

As well as having lists where we know their length before we begin it would be useful to animate the generic case where the length or contents of the list are not important to the result of the program. The animator would instead of iterating over a long list could just perform the operation such as a copy by substituting the body of the list with ellipses. The result of a copy would be the initial list and a new list which also has ellipses denoting that we do not know the length of the list. This would be useful as the heap would not have to be specified before the start of the program. Also this might be useful in programs which perform operations on many lists as it would save space whilst not removing any of its worth as an accurate representation of the heap.

Implementing this from an animation perspective should be an easy enough task. Drawing a generic list would involve drawing one list element at the head of the list followed by ellipses which are then followed by a list element. Operations such as deallocation and allocation of specific list elements would not be possible however deallocation of entire lists would. Operation such as appending lists would be of interest here.

9.3 Hard Coded Heap

The heap (i.e., the programs memory containing the data structures) is hard coded into the system. That means if you wish to change the contents of the heap you need to modify some of the file memory.ml (see Appendix) and recompile the system to use it. If this tool was ever to be used in an educating context this would need revised.

Possible solutions include adding a menu to the system which would allow the user to add or remove list elements to the heap. System-¿Add List Item would bring up a window where the user can specify the data and pointer of the item. If the user tries to make a item point to an item which does not exist the system could ask the user if they would like to create that item, if not then the item would point to NULL.

Another possible solution would be for the system to infer the heap from the logical specification. As this is not the remit for this program it will not be discussed further.

9.4 Reversing the Animation

A nice feature for the user interface would be to allow the user to step back through the animation. To achieve this the system could contain a cache of the recent animations and simply reverse them. After each step the list would update the list by adding a list of operations to the list. Then a button “Previous Step” would look at the last element in the list of animations and reverse it.

The list of steps would need to have a list of animations for each step. This due to the animator having the capability of performing a number of animations in one step. This is done when a data variable is assigned a value, the variable is disposed and a new value is created.
9.5 Summary

Extending the system would be a very useful venture the system could be of great benefit of anyone who wishes to know more about their programs at memory level. The system was developed in such a way that these extensions should be easy to develop. The Position List, Text and Variable classes provide a way of maintaining position of the data and variable elements of the data structures.
Chapter 10

Summary

The final chapter takes a holistic view of the project and comments on the major successes of the project and the development process.

10.1 Successes

The main success of the project was in the way that it was built. Using a generic approach allowed for an incremental development process. This approach allows for a smooth handover for anyone wishing to extend the final system.

The aim of this project was to build a tool which is capable of animating the changes of a program’s data structures during execution. This was achieved successfully as the result of this project is a tool which is capable of animating programs which operate on linked lists.

The way the variables are drawn is a particularly nice feature of this project. It emphasizes the applicable, generic nature of the project by creating the variable objects “on-the-fly”, as and when they are needed. Also the position list which allows for complex layouts and multiple data structures which can change position many times during execution is another nice feature. Having this position list enables the drawing of all animation elements in the same way, i.e., look up the position list to determine their place on the screen.

10.2 Challenges

Obviously there are areas of the project which presented a challenge. I feel that my progress would have been much swifter if I had not had to learn so much at the start of the project. As a novice of functional programming attempting a project of this size was a challenge. My background in object-oriented programming was essential in understanding the classes of OCaml and provided me with a solid foundation for the rest of the project. Also another challenge was learning the LablGtk and LablGL specific interfaces. Having some albeit limited experience with OpenGL these functions were not completely new to me. However how they are implemented in LablGL was. Also I have never programmed in using the GTK+ widget set before. Familiarising myself with all of these tools would be crucial to the overall success of the project.
Also mentioned in Chapter 5 was that a prototype of the system had already been implemented. This proved to be instrumental in helping me successfully build the system. However, as with any new developer working with a new system there is a substantial period of time which has to be spent learning to understand existing code. Fortunately I received exceptional support during the project on all matters relating to OCaml and the original system.

10.3 Opportunities for Improvement

No project is perfect and unsurprisingly there are areas of this project which could be improved. The original intention was to create a system which could animate all functions on a linked list. In its current state it is capable of animating all functions but the Interpreter lacks support for function calls. For that reason a sort function has not been implemented. However the system is capable of handling an insert function which is the first step in building a sort function.

Also the drawing of data is not ideal. The text functions for example only support four characters. Clearly a better way of implementing the text would be to use a bitmapped or texture mapped font. This was not implemented due the concentration of efforts being focussed to the list functions such as allocate and deallocate.

10.4 Summary

This report described the process and motivations which contributing in creating a system capable of animating changes in a programs data structures during execution. The report introduced the solid foundations logic which can be used to prove programs correct[11, 20, 13]. Using these logics enables us to reason about programs at a mathematical level. A significant extension to the logic suggested by Hoare [13] Separation Logic[24] was also discussed as a method of reasoning about programs at a local level. Separation Logic is a useful method for reasoning about pointer programs as its main strength is scalability and locality.

The system built for the topic Animating Pointer Based Data Structures utilises Smallfoot[5] which uses symbolic execution as a method of reasoning about pointer programs [6]. Using Smallfoot this project adds a new level of abstraction, visualisation. The project built a tool for visualising Smallfoot programs focusing solely on singly linked lists. This tool provides an aid to understanding how programs manipulate data structures by animating the changes in a programs heap during the its execution.

The tool supports animation of operations such as fading in, fading out and translating all elements of a linked list including variables.

This project was successful in fulfilling the aims and requirements set out in Chapter ?? and built a tool capable of demonstrating the potential of visualisation software in combination with logical code analysis as a tool which can aid the understanding of our pointer programs.

The main theme of the whole project is how formal verification and static analysis can provide the foundation for visualising the changes to a programs memory. This project introduces a tool which shows only a small fraction of what can be achieved in this field.
Appendix A

OCaml Doc
1 Module Animation: Class to dynamically animate a heap which is updated by an interpreter executing a input program

Author(s): Neil Struth (in part)

class animator : #Memory.memory -> object
   val mutable voi : Memory.var
   val mutable coi : int
   val mutable switch : int
   val mutable cur_t : float * float
   val mutable cur_h : float * float
   val mutable dest_h : float * float
   val mutable dest_t : float * float
   val mutable zoomfac : float

   Amount to zoom

   val mutable alphas : float array

   Transparency of the cons first array entry is used for fading in and second is used for fading out

   val mutable segtrans : float

   Translation amount for list segments

   val mutable dpos : int

   Position of deallocation

   val mutable apos : int

   Position of allocation

   val mutable vars : Variable.variable list
   val mutable pos_print : bool
   val mutable animate_cons : bool
   val mutable al_waiting : bool
   val mutable window_open : bool
   val mutable unfinished_dp : bool
   val mutable unfinished_al : bool
   val mutable allocate : bool
   val mutable initial : bool
   val mutable lr_trans : float
Controls the amount to move the entire animation left and right

val mutable ud_trans : float

Controls the amount to move the entire animation up and down

val mutable cur_store : Memory.store_map list

Current variables and where they point

val mutable next_store : Memory.store_map list

New variables and where they point

val mutable cur_heap :
  (Memory.anim_datastructure * (Memory.var * int) list) list

Copy of the current heap

val mutable next_heap :
  (Memory.anim_datastructure * (Memory.var * int) list) list

Copy of the new heap

val mutable cur_pos : Position.position_lst

The position list

val window : GWindow.window
val appbox : GPack.box
val areabox : GPack.box
val rbox : GPack.box
val gl_area : GlGtk.area
val menubox : GPack.box
val vscrollbox : GPack.box
val hscrollbox : GPack.box
val next_btn : GButton.button
val print_btn : GButton.button
method get_animated : bool
method set_initial : unit -> unit
method initialise_window : unit -> unit

Sets up the window which contains the OpenGL display area and User Interface

method private init_callbacks : GMain.Timeout.id

Initialises callback events for the window
method scroll_cb : string -> unit

Move or scroll the display area

method reset_vars : unit

Reset the fade in, fade out and translate amount and reset animate flag

method update_state : unit -> unit

Updates the state variables. Next store and heap become are updated from memory and current store and heap are updated from next store. Variables are also updated. Heap lists are sorted. Also if this is the first step then initialise the position list

method init_list : unit -> unit

Initialises the position list by passing the most up to date copy of the heap as a parameter to the new position object

method trim_heap :
(Memory.anim_datastructure * (Memory.var * int) list) list ->
Memory.anim_datastructure list

Removes the second "Pointer" elements from the heap

method print_a_cons : Memory.anim_datastructure -> unit

Prints the A_Cons type

method check_rem : int

Compares the current heap and the next heap to determine if a list element has been removed.
Returns the id of the element to be removed

method check_add1 : bool

Compare heaps to determine if a list element has been added
Returns true if a cell has been added else false

method print_store : Memory.store_map list -> unit

Prints the variable store

method initGl : unit -> unit

Sets up the OpenGL display area

method setup : (unit -> unit) -> GtkSignal.id

Sets up the "next_btn" clicked callback. Allows Interface.ml to use its own function for the next_btn.
method reshape : width:int -> height:int -> unit

    Reshape Callback

method get_model :
    unit -> (Memory.anim_datastructure * (Memory.var * int) list) list

    Get the heap from memory

method inc_removed : unit

    Start the deallocation animation

method inc_add : unit -> unit

    Start the allocation animation

method get_var_pos : Memory.var -> Memory.store_map list -> int

    Gets the variable position from the store

method get_vars : Memory.store_map list -> Memory.var list

    Gets a list of all the variables in the store.
    Returns a list of all the variable names in the store

method get_voi : Memory.var
method get_coi : int
method get_switch : int
method set_voi : Memory.var -> unit
method set_coi : Memory.var -> unit
method update : Memory.var -> Memory.var -> unit

    Updates the state information after every step. Called by the interpreter. The checks
    for the allocation and deallocation of list elements are performed here

method get_al_pos : Memory.anim_datastructure list -> int

    Determines the position to allocate the new list element to

method draw_cell : float -> float -> unit
method get_state : Memory.var
method get_oil : unit -> float * float
method get_dest : unit -> (float * float) * (float * float)
method set_dest : float * float -> float * float -> unit
method set_current : float -> float -> unit
method print_oil : float * float -> unit
method move_closer : unit -> unit
Method for animating the pointer movement. Increments x and y positions until they reach their destination specified by cur_h and cur_t.

**Method set_projection : unit -> unit**
Sets up the projection matrix for OpenGL.

**Method draw_ptr : float -> float -> float -> float -> unit**
Draws the arrow (pointer) between list elements.

**Method d_straight : float -> float -> float -> float -> unit**
Draws a straight line between two points in OpenGL.

**Method d_line : float -> float -> float -> float -> unit**
Draws a straight line between two points with an arrow on the end. Used for displaying pointers in the animation.

**Method draw_null_ptr : float -> float -> float -> float -> unit**
Draws a straight line up and down with a box at the bottom to represent the null pointer.

**Method ptr_for_cons : int -> int -> float**
Determines which side of the list element the pointer will stop.

**Method draw_cons : float -> float -> Memory.anim_datastructure * (Memory.var * int) list -> int -> unit**
Draws a list element in the display.

**Method draw_data_struct_seg : int -> int -> bool**
**Method draw_data_struct : bool**

**Method setup_vars : unit -> unit**
Setup all the variables needed for the current step in the animation.

**Method draw_vars : unit**
Draw all the current variables for this step.

**Method draw_static_vars : int * int -> unit**
Draws the variables which will not be animated.

**Method draw_mov_vars : int * int -> unit**
Draw the variables which will be animated.
method draw_v_arr : float -> float -> unit
   Draws the vertical arrow underneath the display variable

method dispose_cons : int -> unit -> unit
   Method to animate the removal of a list element from memory

method allocate_cons : int -> unit
   Method to animate the allocation of a list element in memory

method private finished : int -> bool -> unit
   Removes or allocates a list element to the position list. Needed if the user wishes to skip
   the animation. The method is called if the unfinished flag is set

method idle : unit -> unit
   Display call back function

end

Class to dynamically animate a heap which is updated by an interpreter executing a input
program

Author(s): Neil Struth (in part)

2 Module Position: A class for handling the position of list elements in an OpenGL animation

Author(s): Neil Struth

exception NoCons

class position_lst : Memory.anim_datastructure list ->
   object
      val mutable lst : (int * int * int) list
         The position list - tuples of x and y coordinates

      val mutable hp : Memory.anim_datastructure list
         Local copy of the heap used for initialisation

      method get_lst : (int * int * int) list
         Returns The position list

      method size : int
**Returns** the length of the position list

**method ssort : unit**

Sorts the position list on the ids of the elements in ascending order

**method length : int**

Determines length of the position list

**method set_pos : int -> int -> int -> unit**

Changes the position of id to x,y

**method check_pointed : int -> bool**

Checks if the element is the first element in a list

**method same_level : int -> int -> bool**

Determines if two list elements have the same y position

**method init_pos : (int * int * int) list**

Initialises Position List Assumes that the cons cells are in order. As in list element 1 will be drawn before list element 3 Updates to the position list handled in other functions

**method init_pos1 : (int * int * int) list**

**method get_lst_amnt : int**

Counts the total amount of rows in the position list  
**Returns** the number of rows

**method get_next_y : int**

Determines the next y position for a new list element

**method get_max_x : int -> int**

Determines the next x position for a new list element

**method get_next_pos : int * int**

Gets the x and y position for a new list element  
**Returns** tuple containing the x position followed by the y position

**method get_cons_pos : int -> int * int**

Gets the position of a list element specified by cons  
**Returns** the position of cons

**method get_id : int -> int**
Given an position this method searches through the list and returns the id which occupies that position

**Returns** the id of the list element in pos

**method remove_cons : int -> unit**
Removes a list element from the position list

**method remove_pos : int -> unit**
Removes a list element from the position list

**method private remat : int -> (int * int * int) list**
**method update_rem : int -> int -> (int * int * int) list**
Moves list elements which are after pos on row y back one place Used when removing a list element in remove_cons

**method find_pos : int -> int**
Calculates the where a list element appears in the position list

**method id_exists : int -> bool**
Checks whether an id exists in the position list

**method add : int -> unit**
Adds a new list element into the position list. The list element will be added at the end of the same row of the last element drawn unless the last element is the 6th element, in which case the new element will be placed at the start of a new row

**method print : unit**
Prints the position list

end

3 Module Text : A class for drawing text on a OpenGL display area

**Author(s):** Neil Struth

**exception CharNotSupported of char**

**class text : float -> float -> object**

**val mutable x : float**
    Position on x axis
val mutable y : float
    Position on y axis

val mutable alpha : float
    Transparency

method set_alpha : float -> unit
method setx : float -> unit
method sety : float -> unit
method setp : float * float -> unit
method getx : float
method gety : float
method private draw_zero : unit -> unit
method private draw_one : unit -> unit
method private draw_two : unit -> unit
method private draw_three : unit -> unit
method private draw_four : unit -> unit
method private draw_five : unit -> unit
method private draw_six : unit -> unit
method private draw_seven : unit -> unit
method private draw_eight : unit -> unit
method private draw_nine : unit -> unit
method private draw_t : unit -> unit
method private draw_o : unit -> unit
method private draw_z : unit -> unit
method private draw_char : char -> unit -> unit
    Takes a character parameter and calls the appropriate method to draw the char in the position specified by instance variables x and y. Called by draw_string
    Raises CharNotSupported when character is not i,o,t or z
    Returns unit

method private draw_digit : int -> unit -> unit
    Draws a single digit on a OpenGL display area
    Returns unit

method draw : int -> unit
    Splits d into tens and units and calls the draw_digit method with tens and units parameters
    Returns unit
method draw_string : string -> unit
   Takes a String parameter s and calls draw_char on each character in the String
   Returns unit

end

4 Module Variable: A class for handling the position, translation
   and alpha transparency of display variables

Author(s): Neil Struth

class variable : Memory.var ->
   object
      val mutable id : Memory.var
         Variables unique string identifier
      val mutable alpha : float
         Transparency
      val mutable opos : int * int
         Original position (x,y)
      val mutable npos : int * int
         New position (x,y)
      val mutable trans : float
         Amount to translate
      val mutable offset : float
         How far right to draw variable. Used when there is more than one variable pointing to
         a single list element

      method get_id : Memory.var
      method get_alpha : float
      method get_opos : int * int
      method get_npos : int * int
      method get_trans : float
      method set_id : Memory.var -> unit
      method set_alpha : float -> unit
method set_opos_x : int -> unit
method set_opos_y : int -> unit
method set_opos : int * int -> unit
method set_npos_x : int -> unit
method set_npos_y : int -> unit
method set_npos : int * int -> unit
method set_trans : float -> unit
method set_offset : float -> unit
method get_pos : int * int

Checks to see if the system is animating, if so then the original position is returned, else the new position is returned

Returns the x,y position of the variable.

method get_distance : int * int

Gets the distance between the original position and the new position

Returns int tuple containing the x and y distance respectively

method get_dpos : float * float

Calculates the drawing position of the variable by multiplying the position by the element width and adding the translation and offset. Height is calculated by multiplying the position by minus the height of a list element

Returns int tuple containing the drawing coordinates for the variable

method get_init_dpos : float * float

Initial dpos is the drawing position using the new position

method inc_alpha : unit -> unit

Increments the opacity

method dec_alpha : unit -> unit

Decrements the opacity

method inc_trans : unit -> unit

Increments the translation amount

method reset_alpha : unit -> unit

Resets opacity to 1

method reset_trans : unit -> unit

Resets trans to 0
method fade_in : unit -> unit
  Recursive method to increase the opacity of the variable until it is completely opaque

method check_fadein : unit -> bool
  Checks to see if the variable will be faded in. If the original position is NULL, and the new position is not NULL then a fade in is needed.
  Returns true if fade in needed, false otherwise

method check_fadeout : unit -> bool
  Checks to see if the variable will be faded in. If the original position is not NULL, and the new position is NULL then a fade out is needed.
  Returns true if fade out needed, false otherwise

method check_move_right : unit -> bool
  Checks whether the variable needs to be moved right
  Returns true if the variable needs moved or false if it doesn’t

method print : unit -> unit
  Prints the contents of the object

end

A class for handling the position, translation and alpha transparency of display variables

Author(s): Neil Struth
Appendix B

Top Level Interaction Diagram
Appendix C

Source Code

C.1 Animation

```plaintext
open Memory
open Text
open Position
open Variable

(* * Class to dynamically animate a heap which is updated by an interpreter
executing a input program
@author Neil Struth (in part)
*)
class animator (mem:#memory) =
    object (self)
        val mutable voi = ""
        val mutable coi = 0
        val mutable switch = 0
        val mutable cur_t = (0..0.)
        val mutable cur_h = (0..0.)
        val mutable dest_h = (0..0.)
        val mutable dest_t = (0..0.)

        (**Amount to zoom*)
        val mutable zoomfac = 1.0

        (** Transparency of the cons first array entry is used for fading in and second
is used for fading out*)
        val mutable alphas = [1.0; 0.0]

        (**Translation amount for list segments*)
        val mutable segtrans = 0.

        (**Position of deallocation*)
        val mutable dpos = -1

        (**Position of allocation*)
        val mutable apos = -1
        val mutable vars = []

        (**---Flags---*)
        val mutable pos_print = false
        val mutable animate_cons = false
        val mutable al_waiting = false
        val mutable window_open = false
        val mutable unfinished_dp = false
        val mutable unfinished_al = false
        val mutable allocate = false
        val mutable initial = true
        (**------------------------*)
```

84
(** Controls the amount to move the entire animation left and right *)
val mutable lr_trans = 0.

(** Controls the amount to move the entire animation up and down *)
val mutable ud_trans = 0.

(** Current variables and where they point *)
val mutable cur_store = mem # get_store

(** New variables and where they point *)
val mutable next_store = mem # get_store

(** Copy of the current heap *)
val mutable cur_heap = []

(** Copy of the new heap *)
val mutable next_heap = []

(** The position list *)
val mutable cur_pos = new position_lst [];

(* Consider making the window into an object *)

(* ----------------------------- Window ----------------------------- *)
val window = GWindow.window ~ border_width : 10 ~ width : 660 ~ height : 320 ~ title : "Animate" ();
val appbox = GPack.vbox ();
val areabox = GPack.hbox ();
val rbox = GPack.vbox ();
val gl_area = GlGtk.area [' DOUBLEBUFFER ; RGBA ; DEPTH_SIZE 16 ; (* 'BUFFER_SIZE bits *) ] ();
val menubox = GPack.hbox ();
val vscrollbox = GPack.vbox ();
val hscrollbox = GPack.hbox ();
val next_btn = GButton.button ~ label : "Next Step" ();
val print_btn = GButton.button ~ label : "Print Stores" ();

(* Consider making the window into an object *)

method get_animated = animate_cons

method set_initial () = if initial then initial <- false

(* Sets up the window which contains the OpenGL display area and User Interface *)
method initialise_window =

(* Set Up Window *)
ignore ( self # reset_vars );
window # allow_grow ;
window # add appbox # coerce ;
appbox # pack menubox # coerce ;
appbox # pack areabox # coerce ;
areabox # pack ~ expand : false ~ fill : true gl_area # coerce ;
let trans_amnt = 5. in
menubox # pack next_btn # coerce ;
(* Uncomment for debugging *)
(* menubox # pack print_btn # coerce ; *)
gl_area # event # add [' KEY_PRESS ];
gl_area # set_size ~ width : 500 ~ height : 250;

(* Zoom In *)
let zoomin_btn = GButton.button ~ label : "ZOOM IN" ~ packing : ( menubox # pack ~ expand : false ) in
zoomin_btn # connect # clicked ~ callback : ( fun () -> gl_area # set_size ~ width : 1000 ~ height : 500 ;
GlDraw.viewport 0 0 1000 500 ;
window # maximize ;
if zoomfac < 1. then
zoomfac <- zoomfac +. 0.1 ;

(* Zoom Out *)

let zoomout_btn = GButton . button ~ label :"ZOOM OUT" ~ packing :( menubox # pack ~ expand :false ) () in
zoomout_btn # connect # clicked ~ callback :( fun () -> if zoomfac > 0. then
  zoomfac <- zoomfac -. 0.1); self # init_callbacks;
(*Window open so set flag to true *) window_open <- true; window # show

(*Initializes callback events for the window*) method private init_callbacks =
  (*Close window and set window_open to false*)
  window # connect # destroy ( fun _ -> window_open <= false; window # destroy ());
  window # event # connect # key_press ~ callback :
  ( fun ev ->
    let key = GdkEvent . Key . keyval ev in
    if key = GdkKeysyms . _Escape then ( window_open <= false; window # destroy (); true )
    else
      if key = GdkKeysyms . _Up then ( self # scroll_cb "up"; true )
      else if key = GdkKeysyms . _Down then ( self # scroll_cb "down"; true )
      else
        if key = GdkKeysyms . _Left then ( self # scroll_cb "left"; true )
        else
          if key = GdkKeysyms . _Right then ( self # scroll_cb "right"; true )
          else false);
  gl_area # connect # realize ~ callback : self # initGl;
  gl_area # connect # display ~ callback : self # idle;
  gl_area # connect # reshape ~ callback : self # reshape;
  print_btn # connect # clicked ~ callback :( fun () ->
    let rec pvars lst =
      match lst with
      | [] -> Printf . printf "}\n"
      | (hd :: tl) -> hd # print; pvars tl
      in pvars vars;
    Printf . printf "cur_store = \n";
    self # print_store cur_store;
    Printf . printf "next_store = \n";
    self # print_store next_store;
    Printf . printf "Var size = %d\n" ( List . length vars);
    cur_pos # print;
    Printf . printf "Number of lists = %d\n" cur_pos # get_lst_amnt;
    flush stdout );
  GMain . Timeout . add ~ms :20 ~ callback :( fun () -> self # idle ();
    if window_open then gl_area # swap_buffers (); true )

  (*Move or scroll the display area
  @param dir the direction to move the display area*) method scroll_cb dir =
    let trans_amnt = 5. in
    match dir with
    | "up" -> if ud_trans >= 0. || zoomfac > 1. then ud_trans <- ud_trans -. trans_amnt
    | "down" -> ud_trans <- ud_trans +. trans_amnt
    | "left" -> lr_trans <- lr_trans +. trans_amnt
    | "right" -> if lr_trans >= 0. || zoomfac > 1. then lr_trans <- lr_trans -. trans_amnt
    | _ -> ()

  (*Reset the fade in, fade out and translate amount and reset animate flag*)
  method reset_vars =
    alphas .(1) <- 0.0;
    alphas .(0) <- 1.0;
    segtrans <- 0.;
    if animate_cons then
      animate_cons <= false;
if allocate then
allocate <- false

(* * Updates the state variables. Next store and heap become are updated from
memory and current store and heap are updated from next store.
Variables are also updated. Heap lists are sorted. Also if this is the first
step then initialise the position list *)

method update_state ()=
  cur_store <- next_store;
  next_store <- mem # get_store;
  self # setup_vars ();
  let comp l1 l2 =
    match fst (l1) with (A_Cons (Ptr (id),_,_,_)) ->
      match fst (l2) with (A_Cons (Ptr (id'),_,_,_)) ->
        id - id' in
    if next_heap != (self # get_model ()) then
      cur_heap <- List.sort comp next_heap;
      next_heap <- List.sort comp (self # get_model ());
    if initial then
      (self # init_list ());

  (* * Initialises the position list by passing the most up to date
  copy of the heap as a parameter to the new position object *)
method init_list () =
  let pos_heap = self # trim_heap next_heap in
  let comp (A_Cons (Ptr (pa),_,_,_)) (A_Cons (Ptr (pb),_,_,_)) =
    pa - pb
  in
    let pos_heap = List.sort comp pos_heap in
    let plst = new position_lst pos_heap in
    cur_pos <- plst;

  (* * Removes the second "Pointer" elements from the heap *)
method trim_heap lst =
  let comp (A_Cons (Ptr (pa),_,_,_)) (A_Cons (Ptr (pb),_,_,_)) =
    pa - pb
  in
    let rec th l nl =
      match l with
      | [] -> nl;
      | (hd::tl) ->
        match (fst hd) with
        | A_Cons (_,_,_,_) -> (th tl ((fst hd)::nl);
      in
        List.sort comp (th lst []);

  (** Removes the A_Cons type
  @param cons the list element to print *)
method print_a_cons cons =
  match cons with
  | A_Cons (Ptr (pin),Data (data),Ptr (pout),s) ->
    Printf.printf "A_Cons (%d,%d,%d,%d)\n" pin data pout s;
  | A_Cons (Null,Data (data),Ptr (pout),s) ->
    Printf.printf "A_Cons(NULL,%d,%d,%d)\n" data pout s;
  | A_Cons (Ptr (pin),Data (data),Null,s) ->
    Printf.printf "A_Cons(%d,%d,NULL,%d)\n" data s;
  | _ ->
    Printf.printf "Unmatched\n";

  (** Compares the current heap and the next heap to determine if
  a list element has been removed.
  @return the id of the element to be removed *)
method check_rem =
  (* Get a list of the ids in the heap *)
let rec idlist lst = match lst with
  | [] -> []
  | (hd::tl) ->
    match hd with
    | A_Cons (Ptr (id),_,_,_) -> id::idlist tl in

  (* Make list of ids in cur_heap and next_heap *)
let cl = idlist (self # trim_heap cur_heap) in
let nl = idlist (self # trim_heap next_heap) in
(* Compare the ids *)

let rec ch l1 l2 =
  match l1 with
  | [] -> -1
  | (hd::tl) -> if List.mem hd l2 then ch tl l2 else hd in

ch cl nl;

(* Compare heaps to determine if a list element has been added
(*@return true if a cell has been added else false*)

method check_add1 =
  (* takes a heap and returns an id list *)
  let rec idlist lst = match lst with
  | [] -> []
  | (hd::tl) -> match hd with
    A_Cons(Ptr(id),___) -> id::idlist tl
  in

  (* Get an id list for each heap *)
  let cl = idlist ( self # trim_heap cur_heap ) in

  let nl = idlist ( self # trim_heap next_heap ) in

  (* Compare the id lists *)
  let rec ch l1 l2 =
  match l1 with
  | [] -> false
  | (hd::tl) -> if List.mem hd l2 then ch tl l2 else true in

ch nl cl;

(* Prints the variable store
(*@param store the variable list to be printed*)

method print_store store = Printf.printf "Store:\n\n"

let rec print l =
  match l with
  | [] -> Printf.printf ("\n"");
  | ((v,Ptr(n)):: tl) -> Printf.printf "%s -> %d\n" v n;
    print tl
  | ((v,Null) :: tl) -> Printf.printf "%s -> Null\n" v;
    print tl
  in

print store;

(* Sets up the OpenGL display area*)

method initGl () =
  GlDraw.shade_model 'smooth;
  GlClear.color ~ alpha:0.0 (0.0, 0.0, 0.0);
  GlClear.depth 1.0;
  Gl.enable 'depth_test;
  GlFunc.depth_func 'lequal;

  Gl.enable 'blend;
  GlFunc.blend_func ~src:src_alpha ~dst:one_minus_src_alpha;
  GlMisc.hint 'perspective_correction 'nicest;
  GlDraw.line_width 0.5

  (** Sets up the "next_btn" clicked callback. Allows Interface.ml to use its own
  function for the next_btn.
  (*@param n the function to be called each time the next button is pressed.*)

method setup n =
  Printf.printf "Next Button Callback Setup:\n"

next_btn#connect#clicked ~ callback:n

(* Reshape Callback*)

method reshape ~ width ~ height =
  print_endline "Reshape called";

let check_height =
  if height = 0 then 1 else height in

let ratio = float_of_int width /. float_of_int check_height in

GlDraw.viewport 0 0 width height;
GlMat.mode 'projection;
GlMat.load_identity ();
GlMat.ortho ~x:(0., 500.) ~y:(0., 250.) ~z:(0., 1.);
GLMat.mode 'modelview;
GLMat.load_identity ();

(**Get the heap from memory*)
method get_model () =
  mem#get_drawn_memory voi coi switch;

(**Start the deallocation animation*)
method inc_removed =
  Printf.printf "Inc Removed\n";
  animate_cons <- true

(**Start the allocation animation*)
method inc_add () =
  let t_heap = (self#trim_heap next_heap) in
  if animate_cons then al_waiting <- true else begin
    al_waiting <- false;
    self#finished apos false;
    unfinished_al <- true;
    allocate <- true;
    Printf.printf "APOS = %d\n" apos
  end

(**Gets the variable position from the store
@param id the variables identifier
@param store the list of variables and their pointers*)
method get_var_pos id store =
  let rec gvp ident s =
    match s with
      | [] -> -1
      | (hd :: tl) -> (match (fst hd) with
        | var -> (if ((compare var ident) = 0) then (match (snd hd) with
          | Null -> -1
          | Ptr(x) -> x)
          else gvp ident tl)
        | _ -> gvp ident tl)
    in gvp id store

(**Gets a list of all the variables in the store.
@param store the list of variables and location information
@return a list of all the variable names in the store*)
method get_vars store =
  let rec gvs s =
    match s with
      | [] -> []
      | (hd :: tl) -> (fst(hd))::gvs tl
    in gvs store

method get_voi = voi
method get_coi = coi
method get_switch = switch

method set_voi v =
  voi <- v; switch <- 0; (Printf.printf "voi = %s\n" voi)
method set_coi v =
  match ((mem#get_index v)) with
    | Memory.Null ->
      coi <- -1
    | Ptr(nn) ->
      coi <- nn;
  if ((mem#fld_lookup_mem v "tl") != -1)
    then (switch <- 1; (Printf.printf "coi = %d\n" coi);
        let pos = cur_pos#get_cons_pos(mem#fld_lookup_mem v "{tl}") in
        cur_t <- (float_of_int (((nn) * 80) + 10), 90.);
        cur_h <- (float_of_int (((fst(pos)) * 80) + 10), 140.);
        Printf.printf "cur_h x = %d\n" ((fst(pos) * 80) + 10);
else ((cur_t <- (-1., -1.); cur_h <- (-1.,-1.)))

(* Updates the state information after every step.
 Called by the interpreter. The checks for the allocation and
 deallocation of list elements are performed here *)

method update v1 v2 =
  self # update_state ();
  let disp = self # check_rem in
  Printf.printf "cur_heap length = %d\nnext_heap length = %d\n" (List.length cur_heap) (List.length next_heap);
  if (disp > -1) then (Printf.printf "DEALLOCATE\n"; dpos <- disp; self # inc_removed);
  if(self # check_add1) then (Printf.printf "ALLOCATE\n"; self # inc_add ());
  if (not (v1 = "_nil_"))
    then self # set_voi v1
    else self # set_coi v2

(* Determines the position to allocate the new list element to *)

method get_al_pos lst =
  let comp (A_Cons (Ptr (pa),_,_,_)) (A_Cons (Ptr (pb),_,_,_)) =
    pb - pa in
  let nheap = List.sort comp lst in
  match List.hd nheap with A_Cons (Ptr (id),_,_,_) -> id

method draw_cell x y =
  GlMat.push();
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x) ~y:(y *. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y) ();
  GlDraw.ends();
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y *. 60.) ();
  GlDraw.vertex ~x:(x) ~y:(y *. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.ends();
  GlMat.pop ()

method get_state =
  self # get_voi

method get_oi () = cur_h

method get_dest () =
  (dest_h, dest_t)

method set_dest h t = dest_h <- h; dest_t <- t(*; Printf.printf "dest: (%3f,%3f) (%3f,%3f)\n" (fst(h)) (snd(h)) (fst(t)) (snd(t))*)

method set_current x y =
  cur_h <- (x,y)

method print_oi oi =
  Printf.printf "oi-> x = %f y = %f" (fst oi) (snd oi);
  flush stdout

(* Method for animating the pointer movement. Increments x and y positions until
 they reach their destination specified by cur_h and cur_t *)

method move_closer () =
  let dx = 1. in
  let dy = 1. in
  if ((fst(cur_h)) > (fst(dest_h)))
    then if ((snd(cur_h)) > (fst(dest_h)))
then cur_h <- (((fst cur_h) -. dx), ((snd cur_h) -. dy))
else cur_h <- (((fst cur_h) -. dx), ((snd cur_h) +. dy))

if ((snd(cur_h)) > (fst(dest_h))) then cur_h <- (((fst cur_h) +. dx), ((snd cur_h) -. dy))
else cur_h <- (((fst cur_h) +. dx), ((snd cur_h) +. dy))

(* * Sets up the projection matrix for OpenGL *)
method set_projection () =
  GlMat.mode 'projection;
  GlMat.load_identity ();
  GlMat.ortho ~x:((0., 500.)) ~y:((0., 250.)) ~z:((0., 1.));
  GlMat.mode 'modelview;
  GlMat.load_identity ()

(* * Draws the arrow (pointer) between list elements *)
method draw_ptr xo yo xd yd =
  GlDraw.begins 'line_strip;
  self#d_line xo yo xd yd;
  GlDraw.ends ()
  GlDraw.begins 'quads;
  GlDraw.vertex ~x:(xo -. 2.) ~y:(yo -. 2.) ();
  GlDraw.vertex ~x:(xo -. 2.) ~y:(yo +. 2.) ();
  GlDraw.vertex ~x:(xo +. 2.) ~y:(yo +. 2.) ();
  GlDraw.vertex ~x:(xo +. 2.) ~y:(yo -. 2.) ();
  GlDraw.ends ()

(* * Draws a straight line between two points in OpenGL *)
method d_straight xo yo xd yd =
  GlDraw.vertex ~x:(xo) ~y:(yo) ();
  GlDraw.vertex ~x:(xd) ~y:(yd) ();
  GlDraw.vertex ~x:(xd +. 5.) ~y:(yd -. 5.) ();
  GlDraw.vertex ~x:(xd) ~y:(yd) ();
  GlDraw.vertex ~x:(xd -. 5.) ~y:(yd -. 5.) ();

(* * Draws a straight line between two points with an arrow on the end. Used for displaying pointers in the animation *)
method d_line xo yo xd yd =
  let cx = xd -. xo in
  let cy = yd -. yo in
  if(cx == 0.) then self#d_straight xo yo xd yd else
    begin
      let m = cy /. cx in
      let c = yo -. (m *. xo) in
      let chigh = c +. 10. in
      let clow = c -. 10. in
      let xhigh = (yd -. chigh) /. m in
      let ylow = (m *. xd) +. clow in
      GlDraw.vertex ~x:(xhigh) ~y:(yd) ();
      GlDraw.vertex ~x:(xd) ~y:(ylow) ();
      GlDraw.vertex ~x:(xd) ~y:(yd) ();
      GlDraw.vertex ~x:(xd -. 10.) ~y:(yd -. 10.) ();
    end

(* * Draws a straight line up and down with a box at the bottom to represent the null pointer *)
method draw_null_ptr xo yo xd yd =
  GlDraw.begins 'line_strip;
  GlDraw.vertex ~x:(xo) ~y:(yo) ();
  GlDraw.vertex ~x:(xd) ~y:(yd) ();
  GlDraw.ends ()
  GlDraw.begins 'quads;
  GlDraw.vertex ~x:(xo -. 2.) ~y:(yo -. 2.) ();
  GlDraw.vertex ~x:(xo -. 2.) ~y:(yo +. 2.) ();
  GlDraw.vertex ~x:(xo +. 2.) ~y:(yo +. 2.) ();
  GlDraw.vertex ~x:(xo +. 2.) ~y:(yo -. 2.) ();
  GlDraw.ends ();
  GlDraw.begins 'quads;
  GlDraw.vertex ~x:(xd -. 10.) ~y:(yd -. 10.) ();
(** Determines which side of the list element the pointer will stop *)

method ptr_for_cons n a =
  if(a < n) then float_of_int (((n) * 80) + 10)
  else float_of_int (((n) * 80) + 70)

(** Draws a list element in the display. *)

method draw_cons x y m c =
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y) ();
  GlDraw.ends();
  GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y +. 60.) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 120.) ();
  GlDraw.vertex ~x:(x +. 60.) ~y:(y +. 60.) ();
  GlDraw.ends();
  match (fst m) with
  | A_Cons(_ptrin,h,t,s) ->
    (* Examine the A_Cons and display the data *)
    match h with
    | Data(d) -> let text = new text (x +. 25.) (y +. 80.) in
      text # draw d;
    if (s == 1) then
      match t with
      | Memory.Null ->
        self # set_dest ((x +. 30.), (y +. 30.) ((x +. 30.), (120.));
        let sit = self # get_oi () in
        self # draw_ptr (x +. 30.) (y +. 30.) (fst(sit)) snd(sit);
        self # move_closer ()
      | Ptr(tt) ->
        let postt = (fst (cur_pos # get_cons_pos tt)) in
        let yp = (snd (cur_pos # get_cons_pos tt)) in
        let ypos = (float_of_int (yp) *. 180.) in
        let posc = (fst (cur_pos # get_cons_pos c)) in
        self # set_dest ((self # ptr_for_cons postt posc), (y +. 90.) ((x +. 30.), (y +. 30.))
        (if ((fst(sit)) < 0.) then (self # set_current (x +. 30.) (120.));
        self # draw_ptr (x +. 30.) (y +. 30.) (fst(sit)) (150.);
        self # move_closer ()
      )
    else
      (match t with
       | Memory.Null ->
        self # draw_null_ptr (x +. 30.) (y +. 30.) (x +. 30.) (y -. 30.)
       | Ptr(tt) ->
        let postt = (fst (cur_pos # get_cons_pos tt)) in
        let yp = (snd (cur_pos # get_cons_pos tt)) in
        let ypos = (float_of_int (yp) *. 180.) in
        let posc = (fst (cur_pos # get_cons_pos c)) in
        self # draw_ptr (x +. 30.) (y +. 30.) (self # ptr_for_cons postt posc)
        (if (not (cur_pos # same_level postt posc)) then ((y +. 90.) -. (float_of_int (yp) *. 180.))
          else (y +. 90.))
      )
    )
  else
    (match t with
     | Memory.Null ->
      self # draw_null_ptr (x +. 30.) (y +. 30.) (x +. 30.) (y +. 30.)
     | Ptr(tt) ->
      let postt = (fst (cur_pos # get_cons_pos tt)) in
      let yp = (snd (cur_pos # get_cons_pos tt)) in
      let ypos = (float_of_int (yp) *. 180.) in
      let posc = (fst (cur_pos # get_cons_pos c)) in
      self # draw_ptr (x +. 30.) (y +. 30.) (self # ptr_for_cons postt posc)
      (if (not (cur_pos # same_level postt posc)) then ((y +. 90.) -. (float_of_int (yp) *. 180.))
        else (y +. 90.))
    )
(* Draw a segment of the data structure used for animation only *)
method draw_data_struct_seg start fin =
  let heap = if allocate then next_heap else cur_heap in
  let y = 60. in
  let rec ddseg lst con =
    let id = cur_pos # get_id con in
    let ypos = y +. (float_of_int (snd (cur_pos # get_cons_pos id)) *. -180.) in
    let xpos = (10. +. (float_of_int (fst (cur_pos # get_cons_pos id)) *. 80.) ) in
    match lst with
    | [] -> true
    | (hd::tl) -> if(con <= fin) then
      (if(con < start) then
        ignore (ddseg tl (con + 1))
      else self # draw_cons xpos ypos hd id; ddseg tl (con + 1) ;
      )
    else true;
  in ddseg heap
(* self # get_model () *)

(* Draw entire data structure used when not animating *)
method draw_data_struct =
  let y = 60. in
  let rec dds lst con =
    let id = cur_pos # get_id con in
    let pos = cur_pos # get_cons_pos id in
    let ypos = y +. (float_of_int (snd (pos )) *. -180.) in
    let xpos = (10. +. (float_of_int(fst(pos )) *. 80.) ) in
    match lst with
    | [] -> true
    | (hd::tl) -> if pos = (-1,-1) then (dds tl (con + 1)) else (self # draw_cons
      xpos ypos hd id; dds tl (con + 1))
  in dds (self # get_model () *) 0

(* Setup the all the variables needed for the current step in the animation *)
method setup_vars () =
  (* Create the variables *)
  let nvars = self # get_vars next_store in
  Printf.printf "nvars size = %d\n" (List.length nvars);
  let rec sp lst =
    match lst with
    | [] -> []
    | (hd::tl) -> new variable hd::sp tl
  in vars <- sp nvars ;
  (* Set their new positions *)
  let rec set_npos lst =
    match lst with
    | [] -> ()
    | (hd::tl) -> hd # set_npos (cur_pos # get_cons_pos (self # get_var_pos (hd # get_id)
      next_store)); set_npos tl
  in set_npos vars ;
  (* Set their original positions *)
  let rec set_opos lst =
    match lst with
    | [] -> ()
    | (hd::tl) -> hd # set_opos (cur_pos # get_cons_pos (self # get_var_pos (hd # get_id)
      cur_store)); set_opos tl
  in set_opos vars ;

  (* Draw all the current variables for this step *)
  method draw_vars =
    let rec draw lst n =
      (* For each variable drawn the offset increases by 5 *)
      let off = 5. *. float_of_int(n) in
      let t = new text 0. 0. in
      match lst with
      | [] -> ()
      | (hd::tl) -> hd # draw_cons (t off (xpos ypos hd id))
      in draw tl (n+1)
672 | [ ] -> ()
673 | (hd::tl) -> (  
674     if(hd#check_fadein ()) then  
675         (hd#reset_alpha (); hd#inc_alpha ());  
676     if(hd#check_fadeout ()) then  
677         (hd#dec_alpha ());  
678     if(hd#check_move_right ()) then  
679         (hd#inc_trans ());  
680         hd#set_offset off;  
681         t#set_alpha hd#get_alpha;  
682         t#setp hd#get_dpos;  
683         t#draw_string hd#get_id;  
684         self#draw_v_arr t#getx t#gety; draw tl (n+1))  
685     in draw vars 0  
686     687     (**Draw the variables which will not be animated  
688     @param pos the position of the list element being deallocated*)  
689     method draw_static_vars pos =  
690     let rec draw lst n =  
691     let off = 5. *. float_of_int(n) in  
692     let t = new text 0. 0. in  
693     match lst with  
694     | [ ] -> ()  
695     | (hd::tl) -> (  
696         (**If the variable is on the same line and is further right  
697         than pos then do not draw it*)  
698         if (snd(pos) = snd(hd#get_npos)) && (fst(hd#get_npos) >= fst(pos)) then draw tl n else(  
699             if(hd#check_fadein ()) then  
700                 (hd#reset_alpha (); hd#inc_alpha ());  
701             if(hd#check_fadeout ()) then  
702                 (hd#dec_alpha ());  
703             if(hd#check_move_right ()) then  
704                 (hd#inc_trans ());  
705                 hd#set_offset off;  
706                 t#set_alpha hd#get_alpha;  
707                 t#setp hd#get_dpos;  
708                 t#draw_string hd#get_id;  
709                 self#draw_v_arr t#getx t#gety; draw tl (n+1))  
710             in draw vars 0  
711     )  
712     610     (**Draw the variables which will be animated  
713     @param pos the position of the list element being deallocated*)  
714     method draw_mov_vars pos =  
715     let rec draw lst n =  
716     let off = 5. *. float_of_int(n) in  
717     let t = new text 0. 0. in  
718     match lst with  
719     | [ ] -> ()  
720     | (hd::tl) -> (  
721         (**If the variable is not on the same line or is further left  
722         than pos then do not draw it*)  
723         if (snd(pos) = snd(hd#get_npos)) && (fst(pos) > fst(hd#get_npos)) then(  
724             if(hd#check_fadein ()) then  
725                 (hd#reset_alpha (); hd#inc_alpha ());  
726             if(hd#check_fadeout ()) then  
727                 (hd#dec_alpha ());  
728             if(hd#check_move_right ()) then  
729                 (hd#inc_trans ());  
730                 hd#set_offset off;  
731                 t#set_alpha hd#get_alpha;  
732                 t#setp hd#get_dpos;  
733                 t#draw_string hd#get_id;  
734                 self#draw_v_arr t#getx t#gety; draw tl (n+1)) else draw tl n  
735             in draw vars 0  
736     )  
737     613     (**Draws the vertical arrow underneath the display variable  
738     @param x the x position  
739     @param y the y position*)  
740     method draw_v_arr x y =  
741     GlDraw . begins ' line_strip ;  
742     GlDraw . vertex ~x:(x +. 30.) ~y:( y +. 150.) ();  
743     GlDraw . vertex ~x:(x +. 30.) ~y:( y +. 120.) ();
GlDraw.ends ();
GlDraw.begins 'line_strip;
    GlDraw.vertex "x:(x *+. 25.) y:(y *+. 135.) ()";
    GlDraw.vertex "x:(x *+. 30.) y:(y *+. 120.) ()";
GlDraw.ends ();
GlDraw.begins 'line_strip;
    GlDraw.vertex "x:(x *+. 35.) y:(y *+. 135.) ()";
    GlDraw.vertex "x:(x *+. 30.) y:(y *+. 120.) ()";
GlDraw.ends (

(**Method to animate the removal of a list element from memory
@param id the integer id of the element which is being removed*)
method dispose_cons id =
    (**Since the animation has started it is not fnsihed*)
    unfinished_dp <- true;

    (**Get the position of the list element*)
    let pos = cur_pos # find_pos id in

    (**Draw the data structure up to the point of deallocation*)
    self # draw_data_struct_seg 0 (pos - 1);
    let vpos = cur_pos # get_cons_pos id in
    self # draw_static_vars vpos ;

    (**Set the color to opaque*)
    GlDraw.color ~ alpha : alphas .(0) (0.0 , 0.0 , 0.0) ;

    (**The first part of the animation
    is finished if the list element is completely transparent*)
    let finished = if (alphas .(0) <= 0.) then true else false in

    (** Draw the list element to remove*)
    self # draw_data_struct_seg pos (pos);

    (** Reset the color back to opaque*)
    GlDraw.color ~ alpha :1.0 (0.0 , 0.0 , 0.0) ;

    (**If the element has completely faded out then start moving the rest of the list
    left
    else draw the rest of the data structure*)
    if( finished && segtrans > -80.) then
    { segtrans <- segtrans -. 1. ;
        GlMat.push ;
        GlMat.translate "x:segtrans "y:0. ();
    }
    self # draw_data_struct_seg (pos + 1) (cur_pos # size);
    GlMat.pop ;
    self # draw_mov_vars vpos ;

    (**If animation finished then set flag to false and
    switch the position list to the current positions --
    changed only one position list which is updated after an animation has taken
    place*)
    if segtrans <= -80. then
    self # finished dpos true ;
    GlMat.push

(**Method to animate the allocation of a list element in memory
@param id the integer identifier of the list element*)
method private allocate_cons id =
    (**Animation has started so it is unfinished*)
    unfinished_al <- true ;
    apos <- id ;

    (**Draw the data structure up to the position we wish to allocate*)
    self # draw_data_struct_seg 0 (id - 1) ;

    (**set color to completely transparent and draw element*)
    GlDraw.color ~ alpha :alphas .(1) (0.0 , 0.0 , 0.0) ;
self#draw_data_struct_segment (id) (id);

(* If element completely opaque then animation has finished *)
if alphas.(1) >= 1. then (allocate <= false);
GlMat.pop;
self#draw_vars

(* Removes or allocates a list element to the position list *n)
Needed if the user wishes to skip the animation. The method is called if the
unfinished flag is set
@param pos the position to allocate or deallocate
@param disp true if disposing, false if allocating*)
method private finished pos disp =
if disp then
begin
unfinished_dp <- false;
(* Printf.printf "Removed - %d" pos;*)
anim_con <- false;
self#setup_vars ();
if al_waiting then
  self#inc_add ()
end
else
begin
cur_pos#add pos;
selsetup_vars ();
end

(* Display call back function*)
method idle () =
  self#set_projection ();
  GlClear.color ~alpha:1. (1.0, 1.0, 1.0);
  GlClear.clear [' color ];
  GlDraw.color ~alpha:1. (0.0, 0.0, 0.0);
  (* Zoom *)
  GlMat.scale ~x: (zoomfac) ~y: (zoomfac) ();
  (* Scroll *)
  GlMat.translate ~x: lr_trans ~y: ud_trans ();
  (* Decrement the fade out alpha value if the animate_con flag is set*)
  if animate_con then
    begin
      self#dispose_con dpos;
      ( if (alphas.(0) > 0.) then
        alphas.(0) <- alphas.(0) -. 0.02)
    end
  else
    (* Finish of the dispose operation if the unfinished flag is set*)
    if unfinished_dp then
      self#finished dpos true ;)
  (* If allocate flag is set then increase the fade in value and animate*)
  if allocate then
    ( self#allocate_con apos;
      if (alphas.(1) < 1.) then
        alphas.(1) <- alphas.(1) +. 0.01
      else allocate <- false ;)
    (* If no animation needed then draw data structure and variables*)
    if not(animate_con || allocate) then
      ( ignore(self#draw_data_struct);
        self#draw_vars);
      Gl.flush;
      flush stdout;
      if(window_open) then gl_area#swap_buffers ()
    (* Updates the state at the start*)
C.2 Position

(* *A class for handling the position of list elements in an OpenGL animation*)

open Memory
exception NoCons;;

class position_lst (heap :(anim_datastructure) list) =
  object (self)
  (* *The position list - tuples of x and y coordinates*)
  val mutable lst = []

  (* *Local copy of the heap used for initialisation*)
  val mutable hp = []

  (**@return The position list*)
  method get_lst = lst

  (**@return the length of the position list*)
  method size = List.length lst

  (**Sorts the position list on the ids of the elements in ascending order*)
  method ssort =
  let comp (i,_,_) (ii ,_,_) =
    i - ii
  in
  lst <- List.sort comp lst

  (**Determines length of the position list*)
  method length =
  let rec len lis n =
    match lis with
    [] -> n
    | h::t -> len t (n+1)
  in len lst 0

  (**Changes the position of id to x ,y*)
  @param id the id of the list element
  @param x the new x position of the list element
  @param y the new y position of the list element
  *)
  method set_pos (id:int) x y =
  let rec sp l nl =
    match l with
    [] -> nl
    | (hd::tl) -> match hd with
      (i,xx,yy) -> if i = id then (id,x,y)::(sp tl nl) else (i,xx ,yy)::(sp tl nl)
  in lst <- sp lst []

  (**Checks if the element is the first element in a list*)
  @param cons the list element which is checked*)
  method check_pointed cons =
let rec cpoint lst =  
  match lst with  
  | [] -> false  
  | (hd::tl) -> match hd with  
      | (A_Cons(_,Ptr(ptr),_)) -> if ptr = cons then true else cpoint tl;  
      | (A_Cons(_,Null _)) -> cpoint tl;  
  in cpoint hp

(* * Determines if two list elements have the same y position  
  @param c1 the first list element  
  @param c2 the second list element *)  
method same_level c1 c2 =  
  let y1 = snd (self # get_cons_pos c1) in  
  let y2 = snd (self # get_cons_pos c2) in  
  if y1 = y2 then true else false

(* * Initialises Position List  
Assumes that the cons cells are in order.  
As in list element 1 will be drawn before list element 3  
Updates to the position list handled in other functions *)  
method init_pos =  
  let rec ipos lst x y nl =  
    match lst with  
    | [] -> List.rev nl;  
    | A_Cons(Ptr(pin),_,_,_)::tl ->  
      let same_list = self # check_pointed pin || x = 0 in  
      let ypos = if same_list then 0 else 1 in  
      if not same_list then (Printf.printf "New List\n"; ipos tl 1 (y + 1) (pin,0,(y + 1)::nl))  
      else ipos tl (x + 1) y ((pin,x,y)::nl);  
  in ipos hp 0 0 []

method init_pos1 =  
  let rec ipos lis x y n =  
    match lis with  
    | [] -> []  
    | A_Cons(Ptr(pin),_,_,_)::tl ->  
      if n > 5 then (pin,(0),(y + 1))::ipos tl 0 (y + 1) (0)  
      else (pin,(x + 1),y)::ipos tl (x + 1) y (n + 1)  
  in ipos hp (-1) 0 0

(* * Counts the total amount of rows in the position list  
@return the number of rows *)  
method get_lst_amnt =  
  let rec gno lis cnt =  
    match lis with  
    | [] -> cnt  
    | (hd::tl) ->  
      match hd with  
      | (id,x,y) -> if y >= cnt then gno tl (cnt + 1)  
      else gno tl cnt  
      | _ -> cnt  
  in gno lst 0

(* * Determines the next y position for a new list element *)  
method get_next_y =  
  let rec gny lis =  
    match lis with  
    | [] -> -1  
    | (hd::tl) -> match hd with  
      | (_,x,y) -> if (self # get_max_x y) < 5 then y else gny tl  
      in gny lst  

(* * Determines the next x position for a new list element *)
method get_max_x l =
  let rec gleft lis high =
    match lis with
    | [] -> high
    | (hd :: tl) -> match hd with
      | (id, x, y) -> if y = l then
        if x < high then gleft tl high else gleft tl x
        else gleft tl high
    in gleft lst (-1)

(* * Gets the x and y position for a new list element *)
method get_next_pos = ((self#get_max_x (self#get_next_y)), (self#get_next_y))

(* * Gets the position of a list element specified by cons
 @param cons the id of the list element
 @return the position of cons *)
method get_cons_pos cons =
  let rec gcp lis =
    match lis with
    | [] -> (-1, -1)
    | (hd :: tl) -> match hd with
      (id, x, y) -> if id = cons then (x,y) else gcp tl
    in gcp lst

(* * Given a position this method searches through the list and returns the id
 which occupies that position
 @param pos a tuple containing the x and y positions for a list element
 @return the id of the list element in pos*)
method get_id pos =
  let rec get lis n =
    match lis with
    | [] -> -1
    | (hd :: tl) ->
      match hd with
      | (id, _, _) -> if n = pos then id else get tl (n + 1)
    in get lst 0

(* * Removes a list element from the position list
 @param id the id of the list element*)
method remove_cons id =
  if id != -1 then (let coord = self#get_cons_pos id in
    let p = self#find_pos id in
    Printf.printf "Removed %d\n" p;
    if p >= 0 then
      begin
        lst <- self#remat p;
        lst <- self#update_rem p (snd(coord))
      end)

(* * Removes a list element from the position list
 @param pos a tuple containing the x and y position of the list element*)
method remove_pos pos =
  let id = self#get_id pos in
  self#remove_cons id

(* * Moves list elements which are after pos on row y back one place
 Used when removing a list element in remove_cons*)
method private remat pos =
  let rec remove_at l i =
    match l with
    | [] -> []
    | h :: t -> if (i = 0) then t else h :: (remove_at t (i - 1))
  in remove_at lst pos
method update_rem pos y =
  let rec up l n =
    match l with
    [] -> []
  | hd :: tl -> match hd with
    (id,x,yy) -> if (n >= pos && y = yy && x > 0) then (id,(x -1),y)::(up tl (n + 1)) else hd:(up tl (n + 1))
  in up lst 0

(* * Calculates the where a list element appears in the position list
 * @param i the id of the list element*)
method find_pos i =
  let rec find l n =
    match l with
    [] -> -1
  | (hd :: tl) -> match hd with
    (id,_,_) -> if id = i then n else find tl (n + 1)
  in find lst 0

(* * Checks whether an id exists in the position list
 * @param id the list element identifier*)
method id_exists id =
  let rec idlist lis = match lis with
    [] -> []
  | (hd :: tl) -> match hd with
    ((id),_,_) -> id :: idlist tl in
  let ilst = idlist lst in
  if List.mem id ilst then true else false

(* * Adds a new list element into the position list. The list element will be
 added at
 the end of the same row of the last element drawn unless the last element is
 the 6th
 element, in which case the new element will be placed at the start of a new
 row
 * @param id the list elements identifier*)
method add id =
  if self#id_exists id then () else
  begin
    Printf.printf "Added id -> %d" id;
    let xpos = self#get_max_x n in
    if (xpos < 5) then begin (id,(xpos + 1),n)::lst end else ad (n + 1) in
    lst <- ad 0;
    self#ssort
  end

(* * Prints the position list*)
method print =
  Printf.printf "Position List:\n\n";
  let rec ppos l =
    match l with
    [] -> Printf.printf("\n")
  | ((id,x,y)::tl) -> Printf.printf "\t%d -> x = %d y = %d\n" id x y;
    ppos tl
  in
  ppos lst

initializer (hp <- heap ; lst <- self#init_pos1)

C.3 Text
A class for drawing text on a OpenGL display area

@Author - Neil Struth*

exception CharNotSupported of char;;

class text xpos ypos =
  object (self)
    (** Position on x axis *)
    val mutable x = 0.
    (** Position on y axis *)
    val mutable y = 0.
    (** Transparency *)
    val mutable alpha = 1.
    (*------- Getters and Setters ------*)
    method set_alpha a = alpha <- a
    method setx xv = x <- xv;
    method sety yv = y <- yv;
    method setp xy = x <- fst(xy); y <- snd(xy)
    method getx = x;
    method gety = y;
    (* --------------------------------- *)
    (*------------------ Digits -------------------------------------*)
    method private draw_zero =
      GIMat.push;
      (*GIDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*);
      GIMat.translate "x:0. "y:0. ();
      GIDraw.begins 'line_loop;
      GIDraw.vertex "x:(x) "y:(y) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y +. 15.) ();
      GIDraw.vertex "x:(x) "y:(y +. 15.) ();
      GIDraw.ends ();
      GIMat.pop
    method private draw_one =
      GIMat.push;
      (*GIDraw.color ~alpha:1. (0.0, 0.0, 0.0);*);
      GIMat.translate "x:0. "y:0. ();
      GIDraw.begins 'line_strip;
      GIDraw.vertex "x:(x +. 10.) "y:(y) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y +. 15.) ();
      GIDraw.ends ();
      GIMat.pop
    method private draw_two =
      GIMat.push;
      (*GIDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*);
      GIMat.translate "x:0. "y:0. ();
      GIDraw.begins 'line_strip;
      GIDraw.vertex "x:(x +. 10.) "y:(y) ();
      GIDraw.vertex "x:(x) "y:(y +. 7.5) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y +. 7.5) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y +. 15.) ();
      GIDraw.ends ();
      GIMat.pop
    method private draw_three =
      GIMat.push;
      (*GIDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*);
      GIMat.translate "x:0. "y:0. ();
      GIDraw.begins 'line_strip;
      GIDraw.vertex "x:(x) "y:(y) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y) ();
      GIDraw.vertex "x:(x +. 10.) "y:(y +. 7.5) ();
      GIDraw.vertex "x:(x) "y:(y +. 7.5) ();
      GIDraw.ends ();
      GIMat.pop

101
GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 15.) ();
GlDraw.vertex ~x:(x) ~y:(y +. 15.) ();
GlDraw.ends ();
GlMat.pop

method private draw_four =
GlMat.push;
(* GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*)
GlMat.translate ~x:0. ~y:0. ();
GlDraw.begins 'line_strip;
GlDraw.vertex ~x:(x +. 10.) ~y:(y) ();
GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
GlDraw.ends ();
GlMat.pop

method private draw_five =
GlMat.push;
(* GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*)
GlMat.translate ~x:0. ~y:0. ();
GlDraw.begins 'line_strip;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 15.) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 15.) ();
GlDraw.ends ();
GlMat.pop

method private draw_six =
GlMat.push;
(* GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);*)
GlMat.translate ~x:0. ~y:0. ();
GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
  GlDraw.ends ();
  GlDraw.begins 'line_strip;
  GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 15.) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 15.) ();
GlDraw.ends ();
GlMat.pop

method private draw_seven =
GlMat.push;
GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);
GlMat.translate ~x:0. ~y:0. ();
GlDraw.begins 'line_strip;
  GlDraw.vertex ~x:(x) ~y:(y +. 15.) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 15.) ();
  GlDraw.ends ();
GlMat.pop

method private draw_eight =
GlMat.push;
GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);
GlMat.translate ~x:0. ~y:0. ();
GlDraw.begins 'line_loop;
  GlDraw.vertex ~x:(x) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y) ();
  GlDraw.vertex ~x:(x +. 10.) ~y:(y +. 7.5) ();
  GlDraw.vertex ~x:(x) ~y:(y +. 7.5) ();
GlDraw.ends ();
147 GlDraw.ends();
148 GlDraw.begins 'line_loop;
149 GlDraw.vertex "x: (x) "y: (y + 7.5) ();
150 GlDraw.vertex "x: (x + 10.) "y: (y + 7.5) ();
151 GlDraw.vertex "x: (x + 10.) "y: (y + 15.) ();
152 GlDraw.ends();
153 GlDraw.ends();
154 GlMat.pop

method private draw_nine =
GlMat.push;
158 GlDraw.color "alpha: alpha (0.0, 0.0, 0.0);
159 GlMat.translate "x:0. "y:0. ();
160 GlDraw.begins 'line_strip;
161 GlDraw. vertex "x: (x) "y: (y) ();
162 GlDraw. vertex "x: (x + 10.) "y: (y) ();
163 GlDraw. vertex "x: (x + 10.) "y: (y + 7.5) ();
164 GlDraw.ends();
165 GlDraw.begins 'line_loop;
166 GlDraw. vertex "x: (x) "y: (y + 7.5) ();
167 GlDraw. vertex "x: (x + 10.) "y: (y + 7.5) ();
168 GlDraw. vertex "x: (x + 10.) "y: (y + 15.) ();
169 GlDraw. vertex "x: (x) "y: (y + 15.) ();
170 GlDraw.ends();
171 GlMat.pop

(* ---------------------------------------------------------------- *)
(* ------------------------ Letters --------------------------------- *)
(* ---------------------------------------------------------------- *)

method private draw_i =
176 (* Printf.printf "I drawn alpha = %f" alpha; *)
GlMat.push;
179 GlDraw.color "alpha: alpha (0.0, 0.0, 0.0);
180 GlDraw.begins 'line_strip; (* draw i *)
181 GlDraw. vertex "x: (x + 30.) "y: (y + 160.) ();
182 GlDraw. vertex "x: (x + 30.) "y: (y + 170.) ();
183 GlDraw.ends();
184 GlDraw.begins 'line_strip;
185 GlDraw. vertex "x: (x + 30.) "y: (y + 160.) ();
186 GlDraw. vertex "x: (x + 30.) "y: (y + 170.) ();
187 GlDraw.ends();
188 GlDraw.begins 'line_strip;
189 GlDraw. vertex "x: (x + 30.) "y: (y + 176.) ();
190 GlDraw. vertex "x: (x + 30.) "y: (y + 174.) ();
191 GlDraw.ends();
192 GlMat.pop

method private draw_t =
196 (* Printf.printf "t drawn alpha = %f" alpha; *)
GlMat.push;
199 GlDraw.color "alpha: alpha (0.0, 0.0, 0.0);
200 GlDraw.begins 'line_strip;
201 GlDraw. vertex "x: (x + 30.) "y: (y + 160.) ();
202 GlDraw. vertex "x: (x + 30.) "y: (y + 175.) ();
203 GlDraw.ends();
204 GlDraw.begins 'line_strip;
205 GlDraw. vertex "x: (x + 30.) "y: (y + 160.) ();
206 GlDraw. vertex "x: (x + 30.) "y: (y + 160.) ();
207 GlDraw.ends();
208 GlDraw.begins 'line_strip;
209 GlDraw. vertex "x: (x + 28.) "y: (y + 170.) ();
210 GlDraw. vertex "x: (x + 32.) "y: (y + 170.) ();
211 GlDraw.ends();
212 GlMat.pop

method private draw_o =
217 (* Printf.printf "o drawn alpha = %f" alpha; *)
GlMat.push;
220 GlDraw.color "alpha: alpha (0.0, 0.0, 0.0);
221 GlDraw.begins 'line_strip;
222 GlDraw. vertex "x: (x + 29.) "y: (y + 170.) ();

method private draw_z =
  (* Printf.printf "z drawn alpha = \$f" alpha; *)
GlMat.push;
  GlDraw.color ~alpha:alpha (0.0, 0.0, 0.0);
  GlDraw.begins 'line_strip;
  GlDraw.vertex ~x:(x +. 33.) ~y:(y +. 160.) ();
  GlDraw.vertex ~x:(x +. 28.) ~y:(y +. 160.) ();
  GlDraw.vertex ~x:(x +. 33.) ~y:(y +. 170.) ();
  GlDraw.vertex ~x:(x +. 28.) ~y:(y +. 170.) ();
  GlDraw.ends ();
GlMat.pop;

(* ---------------------------------------------------------------- *)
(* * Takes a character parameter and calls the appropriate *)
(* method to draw the char in the position specified by instance variables *)
(* x and y. *)
(* Called by draw_string *)
(* @param c character which is either i, o, t or z *)
(* @return unit *)
(* @raise CharNotSupported when character is not i, o, t or z *)
method private draw_char c =
  match c with
  | 'i' -> self#draw_i
  | 'o' -> self#draw_o
  | 't' -> self#draw_t
  | 'z' -> self#draw_z
  | other -> raise(CharNotSupported(other))

(* Draws a single digit on a OpenGL display area *)
(* @param d single digit integer to be drawn in OpenGL display area *)
(* @return unit *)
method private draw_digit d =
  match d with
    | 0  -> self#draw_zero
    | 1  -> self#draw_one
    | 2  -> self#draw_two
    | 3  -> self#draw_three
    | 4  -> self#draw_four
    | 5  -> self#draw_five
    | 6  -> self#draw_six
    | 7  -> self#draw_seven
Splits d into tens and units and calls the draw_digit method with tens and units parameters
@param d integer between 0 & 99.
@return unit)

method draw d =
if d >= 0 then ignore(
  if d > 9 then
    begin
      let tens = d / 10 in
      let units = d mod 10 in
      self # draw_digit tens;
      self # draw_digit units
    end
  else self # draw_digit d)

Takes a String parameter s and calls draw_char on each character in the String
@param s String to be drawn on the display area
@return unit *)

method draw_string s =
  let rec dstr ss n =
    if (not (n = (String.length s))) then
      try
        (self # setx (x +. float_of_int(n) *. 5.));
        self # draw_char (s.[n]);
        dstr s (n+1)
    with CharNotSupported(c) -> ();
    in dstr s 0
  in dstr s 0

(** Constructor sets x and y positions of the text depending on the parameters received on construction of the object*)
initializer (self # setx xpos; self # sety ypos)
end;

C.4 Variable

open Memory
(**A class for handling the position, translation and alpha transparency of display variables
@author Neil Struth *)

class variable (i: Memory.var) =
  object (self)
  (**------Instance Variables-----*)

  (**Variables unique string identifier*)
  val mutable id = ""

  (**Transparency*)
  val mutable alpha = 1.

  (**Original position (x,y]*)
  val mutable opos = (0,0)

  (**New Position (x,y]*)
  val mutable npos = (0,0)

  (**Amount to translate*)
  val mutable trans = 0.

  (**How far right to draw variable. Used when there is more than one variable pointing to a single list element*)
  val mutable offset = 0.
  (**-----------------------------*)

/home/neil/animation/text.ml
(*------- Getters ----------*)
method get_id = id
method get_alpha = alpha
method get_opos = opos
method get_npos = npos
method get_trans = trans

(*------- Setters -----------------------------*)
method set_id ar = id <- ar
method set_alpha a = alpha <- a;
  (* Printf . printf " Variable %s Alpha Set to %f\n"
    " id self#set_alpha")
method set_opos_x x = (* if x < 0 then (self#reset_trans ()
    opos <- (x, snd(opos))) *) self#reset_trans ();
  opos <- (x, snd(opos))
method set_opos_y y = (* if y < 0 then (self#reset_alpha ()
    opos <- (fst(opos), y)) *) self#reset_trans ();
  opos <- (fst(opos), y)
method set_opos xy = (* if fst(xy) < 0 || snd(xy) < 0 then (self#reset_alpha ()
    opos <- xy) *) self#reset_trans ();
  opos <- xy
method set_npos_x x = npos <- (x, snd(npos))
method set_npos_y y = npos <- (fst(npos), y)
method set_npos xy = npos <- xy
method set_trans t = trans <- t
method set_offset o = offset <- o

(*-------------------------------------------*)

(* Checks to see if the system is animating, if so then the original position is
  returned, else the new position is returned
@return the x, y position of the variable.*)
method get_pos = if self#check_fadeout () || self#check_move_right () then self
  #get_opos else (self#set_trans 0.; self#get_npos)

(*Gets the distance between the original position and the new position
@return int tuple containing the x and y distance respectively*)
method get_distance = (fst(npos) - fst(opos)),(snd(npos) - snd(opos))

(*Calculates the drawing position of the variable by multiplying
the position by the element width and adding the translation and offset.
Height is calculated by multiplying the position by minus the height of a
list element
@return int tuple containing the drawing coordinates for the variable*)
method get_dpos = (((float_of_int(fst(self#get_pos))) *. 80.) +. trans +.
offset), (60. +. (float_of_int(snd(self#get_pos))) *. -180.))

(*Initial dpos is the drawing position using the new position*)
method get_init_dpos = (((float_of_int(fst(self#get_npos))) *. 80.) +. trans +.
offset), (60. +. (float_of_int(snd(self#get_npos))) *. -180.))

(*Increments the opacity*)
method inc_alpha () = if alpha < 1. then self#set_alpha (alpha +. 0.02) (*;
  Printf . printf " Inc Alpha %f\n"
  alpha*)

(*Decrements the opacity*)
method dec_alpha () = if alpha > 0. then self#set_alpha (alpha -. 0.02)

(*Increments the translation amount*)
method inc_trans () = if trans <= (80. *. float_of_int(snd(self#get_distance)))
  then self#set_trans (trans +. 1.)

(*Resets opacity to 1*)
method reset_alpha () =
  if alpha = 1. then (Printf . printf " Alpha Reset\n"; self#set_alpha 0.)

(*Resets trans to 0*)
method reset_trans () = self#set_trans 0.

(*Recursive method to increase the opacity of the variable until it is
completely opaque*)
method fade_in () =
  Printf . printf " Fade In called\n";
  let rec fin a = self#set_alpha a; if alpha < 1. then fin (a +. 0.002) in
(* * Checks to see if the variable will be faded in. If the original position is NULL, and the new position is not NULL then a fade in is needed. @return true if fade in needed, false otherwise *)
method check_fadein () =
  if opos = ( -1 , -1) && npos != ( -1 , -1) then
    true
  else false
(* * Checks to see if the variable will be faded out. If the original position is not NULL, and the new position is NULL then a fade out is needed. @return true if fade out needed, false otherwise *)
method check_fadeout () =
  if npos = ( -1 , -1) && fst ( opos ) > -1 && snd ( opos ) > -1 then
    true
  else false
(* * Checks whether the variable needs to be moved right @return true if the variable needs moved or false if it doesn’t *)
method check_move_right () =
  if ( fst ( opos ) > -1) && ( fst ( npos ) > -1) && ( fst ( opos )) < ( fst ( npos )) then true
  (* trans <= (80. *. float_of_int ( fst ( npos ))) *)
  else false
(* * Prints the contents of the object *)
method print () =
  Printf.printf
    "Variable %s\n \alpha = %d\n \topos = (%d,%d)\n \tnpos = (%d,%d)\n id alpha (fst(opos)) (snd(opos)) (fst(npos)) (snd(npos))
initializer (self#set_id i; self#print ();)
end;;

C.5 Interpret

Interpret.ml was built as part of the previous system. Only slight modifications were made in this project to include support for if else conditionals and data assignment.
(* return current location *)
method get_location () =
  (match cloc with
   | Some x ->
     x
   | None ->
     Location dummy_loc)

(* evaluate an expression *)
method eval_expr expr =
  let rec evexp exp =
    match exp with
    | (Pexp_num (n)) -> (Int (n))
    | (Pexp_ident (v)) -> (Int (mem # get_ident v))
    | (Pexp_null) -> (Int (-1))
    | (Pexp_infix (op, exp1, exp2)) ->
      match op with
      | "==" -> self # check_eq exp1 exp2
      | "!=" -> (Printf.printf "!="; self # not_eq exp1 exp2)
      | _ -> (Bool (0))
    in (evexp expr)

method eval_exp exp =
  (self # eval_expr (exp . pexp_desc))

(* != *)
method not_eq exp1 exp2 =
  let eq = self # check_eq exp1 exp2
  in
  match eq with
  | (Bool (x)) -> (if (x == 0) then (Bool (1)) else (Bool (0)))
  | _ -> (Bool (0))

(* check equality of two expressions *)
method check_eq exp1 exp2 =
  let e1 = self # eval_exp exp1 in
  let e2 = self # eval_exp exp2 in
  match e1 with
  | (Int (x)) ->
    match e2 with
    | (Int (y)) -> if (x == y) then (Bool (1)) else (Bool (0))
    | (Bool (y)) -> (Bool (0))
  | (Bool (x)) ->
    match e2 with
    | (Int (y)) -> (Bool (0))
    | (Bool (y)) -> if (x == y) then (Bool (1)) else (Bool (0)))

(* checks if another item is in the list *)
method next_itm_possible =
  match cur_itm with
  | [] -> false
  | (hd :: tl) -> true

(* processes the next item *)
method next_itm =
  match cur_itm with
  | [] -> Printf.printf "You shouldn’t see this"
  | (hd :: tl) ->
    match hd with
    | (Pfundecl (fname, params, invariant1, local_decls, statements, invariant2, loc1, loc2)) ->
      (Printf.printf "%s\n" fname;
let rec add_locals lst =
  match lst with
  [] -> ()
  | (hd::tl) ->
      Printf.printf "Adding Local: %s\n" hd;
      mem#addvarnil hd;
      add_locals tl
  in
      add_locals local_decls
); cur_itm <- tl; cur_stmt <- statements

(* checks if another statement is in the list *)
method next_stmt_possible =
  match cur_stmt with
  [] -> false
  | (hd::tl) -> true

method get_var exp =
  match exp.pexp_desc with
  (Pexp_ident (v)) -> v
  | _ -> Printf.printf "no match?\n" x

(* process the next statement *)
method next_stmt =
  match cur_stmt with
  [] -> Printf.printf "You shouldn't see this"
  | (hd::tl) ->
      cloc <- hd.pstm_loc;
      match hd.pstm_desc with
      (Pstm_fcall (s, p)) ->
        (Printf.printf "Function %s Called\n" s; flush stdout)
      | (Pstm_new (v)) ->
        (Printf.printf "New Cons: %s\n" v);
        mem#addcons v;
        anim#update v "_nil_";
        cur_stmt <- tl
      | (Pstm_dispose (n)) ->
        Printf.printf "Dispose:\n"
        (match (n.pexp_desc) with
          (Pexp_ident (v)) ->
            (mem#remcons v; anim#update ""_nil_";))
        cur_stmt <- tl
      | (Pstm_assign (v, n)) ->
        (Printf.printf "Assignment: \n"
        (match (n.pexp_desc) with
          (Pexp_ident (v2)) ->
            (Printf.printf "Assignment Matched\n";
             mem#assign v v2)
          | _ -> (match (self#eval_exp n) with
            (Int(x)) ->
              (if (x == -(1)) then
                (mem#updatetonull v)
              else (mem#updateint v x))
            | _ -> (mem#updateint v (-1)));
            anim#update v "_nil_"; cur_stmt <- tl)
      | (Pstm_fldlookup (v1,v2,c)) ->
        (Printf.printf "Field Lookup: \n"
        (mem#fldlookup v1 (self#get_var v2) c);
        anim#update v1 "_nil_"; cur_stmt <- tl)
      | (Pstm_fldassign (exp1, comp, exp2)) ->
( Printf . printf "Field Assignment:\n";
  ( match mem # get_field_type comp with
    PTR -> ( self # get_var exp1 ); ( mem # fldassign ( self # get_var exp1 ) comp
      ( self # get_var exp2 )
    | _ -> ( match ( self # eval_exp exp1 ) with
        Int(x) -> ( mem # nfldassign ( self # get_var exp1 ) comp x )))
  | ( Pstm # update "" _nil_ ; cur_stmt <- tl ));
  ( Printf . printf "Block:\n";
    cur_stmt <- stms @ tl)
  | ( Pstm # if (exp, stm1, stm2)) ->
    ( Printf . printf "If:\n";
      ( match ( self # eval_exp exp ) with
        ( Bool (x)) ->
          if (x == 1) then
            ( Printf . printf "condition met...\n"; cur_stmt <- [ stm1 ] @ tl)
          else ( Printf . printf "condition not met...\n"; cur_stmt <- [ stm2 ] @ tl))
        | _ -> cur_stmt <- tl)
  | ( Pstm # while (inv, exp, stm)) ->
    ( Printf . printf "While Loop:\n";
      ( match ( self # eval_exp exp ) with
        ( Bool (x)) ->
          if (x == 1) then
            ( Printf . printf "loop entered...\n"; cur_stmt <- [ stm ; hd ] @ tl)
          else cur_stmt <- tl
        | _ -> cur_stmt <- tl))
  | _ -> Printf . printf "No Match...\n"; cur_stmt <- tl ("matching stuff ")
)

(* matching stuff *)

method print_mem =
  mem # print

method next_step =
  if ( self # next_itm_possible == true ) then
    self # next_itm;
    if ( self # next_stmt_possible == true ) then
      self # next_stmt;
      ( mem # get_drawn_memory anim # get_state);
      mem # print
  done

(* run entire program *)

method run_program =
  while ( self # next_itm_possible == true ) do
    self # next_itm;
    while ( self # next_stmt_possible == true ) do
      mem # print;
      self # next_stmt;
      done
  done

(* initial part of the interpreter -- sets up the field types from the top of the
file *)

method start_int lex =

  let ( Pprogram ( comps, items )) = lex in
  ( (* Read the comps fields... *)
    Printf . printf "Adding Fields...\n"
  let rec addfields cmp =
    match cmp with
      [] -> ()
    | ( hd::tl ) ->
      mem # addfield ( List . hd cmp );
      addfields tl
    in
    addfields comps;
    Printf . printf "Executing program...\n";

110
cur_itm <- items;
(*self#run_program;
Printf.printf "Execution complete.\n";
Printf.printf "Memory: ";
mem#print*)
);
end;;

/home/neil/animation/interpret.ml

C.6 Memory

Memory.ml was also built as part of the original system. The majority of the work here was performed by Ewen Maclean. The authors part in this object was to fix some minor bugs to allow the system to manipulate data.

(*Modified! To revert remove all instances of NULL from the symbol list*)

type var = string;;
type ptr = Ptr of int | Null;;
type data = Data of int;;
type offset = int;;
type store_map = (var * ptr);;
type symbol = PTR | INT | NULL;;
type datastructure = Cons of ptr * data * ptr;;
type anim_datastructure = A_Cons of ptr * data * ptr * int;;

class memory =
  object (self)
    val data_struct_size = 9
    val mutable store = ([("i", Ptr (0)) (*;("z", Ptr (6))*)]:(store_map list))

    val mutable heap = ([Cons (Ptr (0), Data (1), Ptr (1)); Cons (Ptr (1), Data (2), Null) (*; Cons (Ptr (2), Data (3), Null)*) (*; Cons (Ptr (3), Data (4), Ptr (4)));
                          Cons (Ptr (4), Data (5), Ptr (5)); Cons (Ptr (5), Data (6), Null)*) (*; Cons (Ptr (6), Data (7), Ptr (7)); Cons (Ptr (7), Data (8), Ptr (8));
                          Cons (Ptr (8), Data (9), Null))]

    val mutable field = ([]:((var * offset) list))

    val mutable level = 0
    val mutable sym = ([("i", PTR)]:((var * symbol) list))
    val mutable fieldtype = ([("hd", INT);("tl", PTR)]:((string * symbol) list))

(* add a field *)
method addfield fld = field <- field @ [(fld, (List.length field))]; Printf.printf "added field: %s\n" fld

(*method print_field =
Printf.printf "Field: {\n";
let rec pfld l =
  match l with
  | [] -> Printf.printf ")\n"
  | (hd::tl) -> match hd with (v, off) -> Printf.printf "(\%s, \%d)\n" v off;
pfld tl;
in pfld field*)

(* get memory location for variable v *)
method get_index v =
  let rec ind v l =
    match l with
    | [] -> Null
    | [v] -> l

47 | (hd::tl) -> if ((compare (fst hd) v) == 0)
48 then (snd hd)
49 else (ind v tl)
50 in
51 ind v store
52
53 method get_store = store;
54
55 (* get list of pointers for drawn memory *)
56 method get_ptrs p voi switch =
57 let rec g_ptrs lst p =
58 (match p with
59 | Null ->
60 (match lst with
61 | [] -> []
62 | _::tl ->
63 g_ptrs tl p)
64 | Ptr (p_p) ->
65 {
66 match lst with
67 | [] -> []
68 | (hd::tl) -> (if (p_p == p)
69 then (if (switch == 0) && ((self # get_index voi) == p)
70 then ((fst hd),0) :: (g_ptrs tl p)
71 else ((fst hd),1) :: (g_ptrs tl p)
72 )
73 else (g_ptrs tl p))
74 | _,Null ->
75 g_ptrs tl p))
76 in g_ptrs store p
77
78 (* get model of memory for drawing *)
79 method get_drawn_memory voi coi switch =
80 let rec dm lst n =
81 (* if (n == data_struct_size) then []
82 else (*
83 match lst with
84 | [] -> []
85 | (Cons (Ptr (pin),d,Ptr (pout))::t2) -> (if ((switch == 1) && (coi == n))
86 then ((A_Cons (Ptr (pin),d,Ptr (pout),1)),self # get_ptrs (Ptr (n)) voi switch) :: (dm t2 (n+1))
87 else ((A_Cons (Ptr (pin),d,Ptr (pout),0)),self # get_ptrs (Ptr (n)) voi switch) :: (dm t2 (n+1))
88 )
89 | (Cons (Ptr (pin),d,Null)::t2) ->
90 ( if ((switch == 1) && (coi == n))
91 then ((A_Cons (Ptr (pin),d,Null,1)),self # get_ptrs (Ptr (n)) voi switch) :: (dm t2 (n+1))
92 else ((A_Cons (Ptr (pin),d,Null,0)),self # get_ptrs (Ptr (n)) voi switch) :: (dm t2 (n+1))
93 )
94 in dm heap 0
95
96 (* get field type *)
97 method get_field_type f =
98 let rec fld (f: string) (l:((string * symbol) list)) =
99 match l with
100 | [] -> INT
101 | (hd::tl) -> (if ((compare (fst hd) f) == 0)
102 then (snd hd)
103 else (fld f tl))
(* get offset for field f *)
method get_field f =
  let rec fld f l =
  match l with
  | [] -> 0
  | (hd::tl) -> if ((compare (fst hd) f) == 0)
  then (snd hd)
  else (fld f tl)
  in
  fld f field

(* ----------- utility functions *)
(* remove a variable from a store_map list -- used by remvar *)
method removevar (v : var) (l : store_map list) =
  Printf.printf "removevar v = %s\n" v;
  let rec rem v l =
  match l with
  | [] -> []
  | (hd::tl) -> if ((compare (fst hd) v) == 0)
  then (rem v tl)
  else (hd::(rem v tl))
  in
  rem v l

(* print out the contents of the store *)
method print_store =
  Printf.printf "Store:\n\n";
  let rec print l =
  match l with
  | [] -> Printf.printf ("\}\n")
  | ((v,Ptr(n))::tl) ->
    Printf.printf ("%d -> %s\n") n v;
    print tl
  | ((v,Null)::tl) ->
    Printf.printf ("%d -> Null\n") v;
    print tl
  in
  print store

(* print out the contents of the heap *)
method print_heap =
  let comp (Cons(Ptr(pa),_,_)) (Cons(Ptr(pb),_,_)) =
    pa - pb
  in
  heap <- List.sort comp heap;
  let rec print l =
  match l with
  | [] -> Printf.printf ("\}\n")
  | Cons(Ptr(pin),Data(d),Ptr(pout))::tl ->
    Printf.printf ("%d -> (Data:%d) %d\n") pin d pout;
    print tl
  | Cons(Ptr(pin),Data(d),Null)::tl ->
    Printf.printf ("%d -> (Data:%d) Null\n") pin d;
    print tl
  in
  print heap

(* print store and heap *)
method print =
  self#print_heap;
  self#print_store;
  self#print_sym;
  (*self#
  print_field;*) flush stdout

(* compare first elements of heap to find highest memory address *)
method getmax l =
  let comp (Cons(Ptr(pa),_,_)) (Cons(Ptr(pb),_,_)) =
    pa - pb
  in
  match l with
  | [] -> -1
  | _ -> (match (List.nth (List.sort comp l) ((List.length l)-1)) with
    | Cons(Ptr(pin),_,_) ->
      pin
    | _ -> -1
    -1)
method getmax_heap = self # getmax heap

(* ------------ modelling functions *)
method inc_level = level <- (level + 1)
method dec_level = level <- (level - 1)

(* ------------ modifying only store *)
method addvar v p = store <- store @ [(v, p)]
method addvarnil v = store <- store @ [(v, Null)]
method remvar (v : var) = store <- (self # removevar v store)
method updatevar v p = self # remvar v; self # addvar v p
method updatetonull v = self # remvar v; self # addvar v Null

(* ------------ modifying only heap *)
method get_mem = self # getmax_heap + 1
method allocate pos len =
  let rec allo pos len =
    match len with
    | 0 -> heap
    | len -> ((Cons(Ptr(pos),Data(-1),Null))::(allo (pos + 1) (len -1))):
      in
    heap <- allo pos len
method remmem pos len =
  let rec rem p l h =
    match h with
    | [] -> []
    | (Cons(Ptr(pin),d,po)::tl) ->
      if ((l > 0) && (pin == p))
      (*then (rem (p) (l) tl)*)
      then
        rem (p + 1) (l -1) tl
      else (Cons(Ptr(pin),d,po))::(rem p l tl)
    in
    match pos with
    | Ptr(pos_p) ->
      heap <- (rem pos_p len heap)
    | Null ->
      ()

(* Frees memory from pos to pos+len *)
method updatemem v n =
  let mp = self # get_index v
  in
  match mp with
  | (Ptr mp_p) ->
    let rec upmem v n l =
      match l with
      | [] -> []
      | (Cons(Ptr(pin),d,po)::tl) ->
        if (pin == mp_p)
        (* This should be more generic for a general datastructure *)
        then
          (Cons(Ptr(pin),Data(n),po))::upmem v n tl
        else
          (Cons(Ptr(pin),d,po))::upmem v n tl
      in
    upmem v n heap
  | Null ->
    []

(* updates memory at variable v to value n - just data here*)
method umem pos f n =
  match pos with
  | Ptr(pos_p) ->
    let rec umem pos n fl l =
      match l with
      | [] -> []
      | Cons(Ptr(pin),d,po)::tl ->
        if (pin == mp_p)
        (* This should be more generic for a general datastructure *)
        then
          (Cons(Ptr(pin),Data(d),po))::umem pos n fl tl
        else
          (Cons(Ptr(pin),d,po))::umem pos n fl tl
      in
    umem pos n heap
  | Null ->
    []
(Cons(Ptr(pin),Data(d),Null)):: upmem pos n fl tl
else
(Cons(Ptr(pin),Data(d),Ptr(n))):: upmem pos n fl tl)
else
(Cons(Ptr(pin),Data(d),Ptr(pout))):: upmem pos n fl tl)
| Cons(Ptr(pin),Data(d),Null):: tl ->
(if (pin == pos_p) then
(* This should be more generic for a general datastructure *)
(if (fl == 0) then
(Cons(Ptr(pin),Data(d),Null)):: upmem pos n fl tl
else
(if (n = (-1))
then
(Cons(Ptr(pin),Data(d),Null)):: upmem pos n fl tl
else
(Cons(Ptr(pin),Data(d),Ptr(n))):: upmem pos n fl tl)
else
(Cons(Ptr(pin),Data(d),Null)):: upmem pos n fl tl)
in
upmem pos n f heap)
| Null ->
[
]

(* set value of a field to destination of a ptr variable *)
method modfield vt f vf=
let nvt = self # get_index vt
in
let nvf = self # get_index vf
in
let nf = self # get_field f
in
if (nf == 1) then
((match nvt with
| Null -> ()
| Ptr(nvt_p) ->
match nvf with
| Ptr(nvf_p) ->
heap <- (self # umem nvt nf nvf_p)
| Null ->
heap <- (self # umem nvt nf (-1))
)
else
failwith "Trying to put a ptr value in a data cell?"

(* set value of a field in a cons to a number *)
method setfield vt f n =
let nvt = self # get_index vt
in
let nf = self # get_field f
in
match nvt with
| Ptr(nvt_p) ->
match nvf with
| Ptr(nvf_p) ->
heap <- (self # umem (Ptr(nvt_p)) nf n)
else
failwith "Trying to put a data value in a pointer cell?"
)
in
| Null ->
()

(* ----- modifying both store and heap *)
method addcons v = self # updatevar v (Ptr(self # get_mem));
self # allocate self # get_mem 1;
self # addsym v PTR

(* removing a cons *)
method remcons v =
self # remmem (self # get_index v) 1;
self # remvar v;
sel# remsym v

115
(* make room for an integer in memory *)
method addint v n =
  self#updatevar v (Ptr (self#get_mem));
  self#allocate self#get_mem 1;
  heap <- self#updatemem v n;
  self#addsym v INT

method remint v =
  self#remmem (self#get_index v) 1;
  self#remvar v;
  self#remsym v

method updateint v n =
  Printf.printf "mem#
updateint
v = %s
n = %d
"
  v n;
  self#remint v;
  self#addint v n;
  self#addsym v INT

method get_int v =
  let rec gint v l =
    match (self#get_index v) with
    | (Ptr v_p) ->
      (match l with
       | [] -> -1
       | (Cons (Ptr (pin), Data (d), pout) :: tl) ->
         (if (v_p == pin)
           then d
           else (gint v tl)))
    | Null -> -1
    in gint v heap

(* ------ modifying symbol table *)
(* add a symbol *)
method addsym v t =
  self#remsym v; sym <- (v, t) :: sym

(* remove a symbol *)
method remsym v =
  let rec rem v l =
    match l with
    | [] -> []
    | (hd :: tl) ->
      if ((compare (fst hd) v) == 0)
        then rem v tl
      else hd :: (rem v tl)
      in
      sym <- rem v sym

(* update a symbol *)
method updatesym v t =
  self#remsym v;
  self#addsym v t

(* look up a symbol in the symbol table *)
method get_sym v =
  let rec getsym v l =
    match l with
    | [] -> NULL
    | (hd :: tl) ->
      if ((compare (fst hd) v) == 0)
        then getsym v tl
      else
        (let (snd hd) = (getsym v tl) in
         (getsym v sym))

(* print symbol table *)
method print_sym =
  Printf.printf "Symbols:
"
  let rec print l =
    match l with
    | [] -> Print ("
    | (hd :: tl) ->
      match (snd hd) with
      | PTR ->
        (Printf.printf "
          \t%s -> %s\n" (fst hd); print tl)
      | INT ->
        (Printf.printf "\t%s -> INT\n" (fst hd); print tl)

116
method fld_lookup_mem v f =
  let rec mem n f ls =
  | [] -> -1
  | (Cons(Ptr(pin),Data(d),Ptr(pout))::tl) ->
    (if ((f == 0) && (pin = n)) then
      d
    else
      (if ((f ==1) && (pin = n)) then
        pout
      else
        mem n f tl)
    | ( Cons (Ptr ( pin ),Data (d),Null ):: tl) ->
      (if ((f == 0) && (pin = n)) then
        d
      else
        (if ((f ==1) && (pin = n)) then
          -1
        else
          mem n f tl))
  in
  match (self # get_index v) with
  | (Ptr pv) ->
    (mem pv (self # get_field f) heap)
  | Null -> -1

(* ----- Interface *)

method get_ident v =
  match (self # get_sym v) with
  | PTR ->
    (match (self # get_index v) with
      | (Ptr pin) ->
        pin
      | Null -> -1)
    | INT -> (self # get_int v)
    | _ -> -1

method assign v1 v2 =
  match (self # get_sym v2) with
  | PTR -> (Printf.printf "Assign ptr %s %s\n" v1 v2; self # updatevar v1 (self # get_index v2); self#addsym v1 PTR)
  | INT -> (Printf.printf "Assign int %s %s\n" v1 v2; self # updateint v1 (self # get_int v2))
  | NULL -> (Printf.printf "Unmatched mem#assign %s %s\n" v1 v2; self#updatetonull v1)

method fldlookup v1 v2 f =
  match (self # get_field_type f) with
  | PTR -> (self#updatevar v1 (Ptr self#fld_lookup_mem v2 f));
  | INT -> (self#updateint v1 (self#fld_lookup_mem v2 f));
  | _ -> (self#updateint v1 (self#fld_lookup_mem v2 f));

method fldassign v1 f v2 =
  self#modfield v1 f v2

method nfldassign v1 f n =
  self#setfield v1 f n
end;

/home/neil/animation/memory.ml
Appendix D

Test Cases

D.1 Append

```c
hd,tl;
list_append(i,z) [list(i) * list(z)] {
    local t;
    if (i == NULL) {
        i = z;
    } else {
        t = i;
        o = t->tl;
        /* list(y) is framed */
        while (o != NULL) [lseg(i,t) * t |-> o * list(o)] {
            t = o;
            o = t->tl;
        }
        t->tl = z;
    }
} [list(i)]
```

D.2 Copy (No Data)

```c
hd,tl;
list_copy(i) [list(i)] {
    local t;
    t = i;
    o = NULL;
    while(t != NULL) [list(o) * lseg(i,t) * list(t)] {
        z = o;
        o = new();
        o->tl = z;
        t = t->tl;
    }
} [list(i) * list(o)]
```

D.3 Copy (Data)
D.4 Reverse

```c
list_reverse(o;i) [data_list(a;i)] {
    local t;
    o = NULL;
    while (i != NULL) [Ex alpha. Ex beta. data_list(alpha;i) * data_list(beta;o) / a = append(reverse(alpha),beta)] {
        t = i->tl;
        i->tl = o;
        o = i;
        i = t;
    }
} [data_list(reverse(a);o)]
```

/home/neil/animation/tests/iter-rev.sf

D.5 Deallocate

```c
list_deallocate(i) [list(i)] {
    local t;
    while(i != NULL) [list(i)] {
        t = i;
        i = i->tl;
        dispose t;
    }
} [emp]
```

/home/neil/animation/tests/deallocate.sf
D.6 Traverse

```c
hd, tl;

list_traverse(i) [list(i)] {
  local t;
  t = i;
  /* lseg(z,t) should be framed */
  while(t != NULL) [lseg(i,t) • list(t)] {
    t = t->tl;
  }
}
```

/home/neil(animation/tests/ldal.sf)

/home/neil(animation/tests/traverse.sf)
Appendix E

User Manual

The user manual is a brief overview of how to use the system.

E.1 System Requirements

To run this system requires some software to be installed.

To run the tool requires the target system to be using a Linux based Operating System which can display 3D OpenGL graphics. The system requires a meagre 1.9Mb of hard disk space and should run on the majority of modern processors.

E.2 Running the Application

To run the application from the terminal, provided that the executable and test data has been extracted a directory:

- Navigate to the directory containing the executable file dsview
- Run the tool using the command: ./dsview

Following the above steps should start the application.

E.3 Using the Tool

Once the application has started you will be greeted with a screen containing the source code for the program which was last used. Writing your own program can be done directly through the text editor or by opening a file containing your program. This is done by either clicking the open button or by navigating through the drop-down menus.
Any changes to the source code can be saved by clicking the **Save** button which will display GNOME style save dialogue. To start the animation simply click **Animate** and the animation window will be displayed.
Once the animation window is open you can advance the animation by pressing the **Next Step** button. The current line being executed is highlighted in the source code interface.

![Screenshot of the source code interface with highlighting](image1)

**Figure E.3:** Screenshot of the source code interface with highlighting

```c
1: hd, tl;
2: /*[list(i.DATA,i)]*/
3: list-copy(2;i) [list(i)] {
4: local t, tt, o, cc;
5: if (i == NULL) (z == NULL);
6: else {
7:     z = new(z);
8:     r = tl = NULL;
9:     t = i = hd;
10:     z = o = t;
11:     tt = i = tl;
12: }
13: /*[exists i1.DATA, o.DATE, i2.DATA, i1.DATA = APPEND i1.DATA (CONS o.DATE i2.DATA) &
14:     lseg(APPEND i1.DATA [w.data], i, tt) * list([i2.DATA, tt]) * lseg([i1.DATA, z, o] * o -> [t:0, hd: o.DATE]) */
15: while (tt != NULL) [lseg(i, tt) * list(tt) * lseg(z, o) * o -> o] {
16:     o = new(o);
17:     oo = tl = NULL;
18:     t = tt = hd;
19:     oo = hd = t;
20:     o = oo;
21:     tt = t = tl;
22: }
23: } [list(i) = list(z)]
24: /*[list(i.DATA,i)]*/
25: /*[list(i.DATA,i)]*/
26: /*[list(i.DATA,i)]*/
```

**Figure E.4:** Animation window showing the **next step** button corresponds to ??

To quit the animation either close the source code interface to shut down the application completely or close the animation window which will allow you to make changes to the source code and re-animate the results.
Bibliography


## List of Figures

1.1 Display variable $i$ .............................................. 5  
1.2 Graphical Representation of Pointers .............................................. 6  
1.3 An example of a linked list taken from [16] .............................................. 6  
2.1 The Axiom of Assignment .............................................. 9  
2.2 The Rule of Composition .............................................. 9  
2.3 The Rule of Iteration .............................................. 10  
2.4 Frame Rule .............................................. 11  
2.5 Function $\text{disp\_tree}$ showing the precondition and post-condition annotations .............................................. 14  
2.6 Proof of $\text{disp\_tree}$ from [5] .............................................. 14  
2.7 Shani’s representation of a binary tree [25] .............................................. 16  
2.8 Table showing the capabilities of OpenGL vs Direct3D .............................................. 19  
2.9 Table showing the results of the comparison of the programming languages .............................................. 22  
4.1 Preliminary design of the user interface .............................................. 26  
4.2 Preliminary design of animation .............................................. 26  
4.3 High level view of the system to be developed .............................................. 29  
5.1 Diagram showing the structure of the original system. .............................................. 31  
5.2 Method to display a list element($\text{cons}$) in the animation .............................................. 32  
5.3 Method to determine which side of the list element the link will point to. .............................................. 32  
5.4 Example of a while loop in use. .............................................. 34  
5.5 Example of a if conditional in use. .............................................. 34  
5.6 Suggestion for specifying an initial heap. .............................................. 34  
6.1 A function to deallocate a list from memory - Taken from Smallfoot examples [5] .............................................. 36  
6.2 A function to copy an existing list - Taken from Smallfoot examples [5] .............................................. 36  
6.3 Method $\text{d\_line}$ taken from animation.ml .............................................. 38  
6.4 Screenshot of the animation before a zoom operation .............................................. 40  
6.5 Screenshot of the animation after a zoom in operation .............................................. 40  
6.6 Screenshot of the animation after a zoom out operation .............................................. 41  
6.7 Screenshot of the animation after a dispose operation .............................................. 41  
6.8 Screenshot of the animation after a new() operation .............................................. 42  
7.1 Method to print a string of text characters in an OpenGL drawing area - taken from text.ml .............................................. 46  
7.2 Method $\text{setup\_vars}$ () from animation.ml, initialises the variables objects as they are needed .............................................. 47  
7.3 $\text{draw\_vars}$ method from animation.ml, draws the display variables .............................................. 48
7.4 Method allocate_cons id taken from animation.ml
7.5 Method to display the variables which are not animated - taken from animation.ml
7.6 Method to display the variables which need to be animated - taken from animation.ml
7.7 Screenshot of the system after the statement t = i;
7.8 Screenshot of the system after the statement o = new();
7.9 Screenshot of the system after the statement t = t -> tl;
7.10 Screenshot of the system after the statement z = o;
7.11 Screenshot of the system after the statement o = new();
7.12 Screenshot of the system after the statement o -> tl = z;
7.13 Screenshot of the system after the statement t = t->tl;
7.14 Initial heap in list_deallocate(i)
7.15 System after the statement dispose t;
7.16 System after the second dispose t;
8.1 Program which copies the structure and contents of a linked list
8.2 Method to print integers in an OpenGL drawing area - taken from text.ml
8.3 Method to print the individual digits in an OpenGL drawing area, called by the method draw - taken from text.ml
8.4 An extract from the next step function in interpret.ml
8.5 A further extract from the next step function in interpret.ml showing the data assignment matching
8.6 Initial heap in copy_list(i)
8.7 State of heap in copy_list(i) after 1 step
8.8 State of heap in copy_list(i) t = i -> hd;
8.9 State of heap in copy_list(i) z->hd = t;
8.10 State of heap in copy_list(i) after oo=new();
8.11 State of heap in copy_list(i) after t=tt->hd;
8.12 Final state of heap in copy_list(i)
E.1 Screenshot of the source code interface
E.2 Save Dialogue
E.3 Screenshot of the source code interface with highlighting
E.4 Animation window showing the next step button corresponds to ??