The Semantics, Formal Correctness and Implementation of History Variables in an Imperative Programming Language

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in the
University of Canterbury
by

Ryan Mallon

Examining Committee

Dr. Tadao Takaoka	Supervisor
Dr. Robert Biddle	External examiner

University of Canterbury
2006

To my family

Acknowledgments

First and foremost I wish to thank Richard Green, without whom this thesis would simply not have been possible. His support and determination in the face of adversity have allowed me to achieve a goal I set myself nearly 20 years ago. I also want to thank my supervisor, Tad, for his patience, guidance, stubbornness (he made me write that) and corny jokes. Again, this thesis could not have existed without his help. Thanks to my Mum and Dad, my brothers Lowell and Kalem, and my sister Nina and her family. Their constant love and support helped me through some of the more difficult times. Thanks also to my Godmother and dear friend Joanne.

Thanks to my fellow occupants of MSCS room 344, both past and present: Oliver, Taher (who passed on the ancient art of submitting a thesis), Jason, Michael, and Steven. Our frequent technical, political and religious discussions, arguments and side projects have made the past 16 months entertaining at least. I would also like to thank the other postgrad students from both the computer science department and the HIT lab. Thanks to my fellow tutors at Canterbury University and everyone that I taught during my three years of tutoring. I learned as much from you as you did from me.

Thanks to my flatmates, Willy and Charlotte, for putting up with me at home, providing coffee breaks, and offering to finish my thesis in purple crayon. Thanks to all my workmates at the Rockycola, Rockpool and Mickey Finn's for keeping me sane on weekends. Thanks to my fencing coach, Victor, who encouraged me to put my education first. Finally thanks to all my friends for their support, and for smiling and nodding whenever I tried to explain what my thesis was about: Brehaut, Jaz (who offered to write me a thesis on chickons [1]), Beebs, Kurt, Ian, Liz and Jason (congratulations on the recent engagement), Karen, Trina, Corey and Heidi, Barry, Anna, Jack, Dangles, Reggie, Matt and anyone else I've missed.

[1] No, it's not spelt wrong.

Abstract

Storing the history of objects in a program is a common task. Web browsers remember which websites we have visited, drawing programs maintain a list of the images we have modified recently and the undo button in a word-processor allows us to go back to a previous state of a document. Maintaining the history of an object in a program has traditionally required programmers either to write specific code for handling the historical data, or to use a library which supports history logging.

We propose that maintaining the history of objects in a program could be simplified by providing support at the language level for storing and manipulating the past versions of objects. History variables are variables in a programming language which store not only their current value, but also the values they have contained in the past. Some existing languages do provide support for history variables. However these languages typically have many limits and restrictions on use of history variables.

In this thesis we discuss a complete implementation of history variables in an imperative programming language. We discuss the semantics of history variables for scalar types, arrays, pointers, strings, and user defined types. We also introduce an additional construct called an "atomic block" which allows us to temporarily suspend the logging of a history variable. Using the mathematical system of Hoare logic we formally prove the correctness of our informal semantics for atomic blocks and each of the history variable types we introduce.

Finally, we develop an experimental language and compiler with support for history variables. The language and compiler allow us to investigate the practical aspects of implementing history variables and to compare the performance of history variables with their non-history counterparts.

Table of Contents

List of Figures

Chapter I

Introduction

In traditional imperative programming languages, a variable maintains only its current value. When we assign to a variable we permanently lose the previous value it contained. If we do want to keep the history of a variable then we need to write specific code, or use a library which provides the necessary functionality, to store the previous values.

1.1 What is a history variable?

We define a history variable as: A variable which maintains a list, or "history", of its previous values in addition to its current value. The number of values stored in a variables history is called its "depth". When a new value is assigned to a history variable the previous value is saved to the history. Any previous value of a history variable can be recalled, but not altered. In the real world (time travel theories [57, 51] aside) we cannot change events that have happened in the past. Likewise, in a programming language we cannot modify the historical values of a history variable. History variables may be simple types such as integers and floating point numbers, or aggregate types such as arrays and structures.

The Fibonacci sequence (0, 1, 1, 2, 3, 5, ...) provides a good example of how history variables can be used. We can use a history variable of depth 1 to easily calculate numbers in the Fibonacci series by adding the current and previous value of the variable together to generate the next number in the sequence. For example, if the history variable has the current value 2 and the previous value 3, then calculating the next number will give the current

value of 3 and a previous value of 2, which can then be added to give the next number 5 in the sequence.

1.2 Why are history variables useful?

Storing and retrieving the history of objects in a computer program is a common task: the undo button in a word processor takes the user back to a previous version of a document, the history file in a web-browser stores a list of the previous websites a user has visited and the auto-fill option present in some applications remembers the previous entries typed by a user. We discuss the applications of history storage in computing in more detail in section 2.1.

All of the functionality provided by history variables can already be accomplished using existing languages and libraries. Our goal in implementing history variables as language feature is to provide a simplified approach to the common task of storing the history of objects in a computer program. For example, the Eclipse [26] development environment has around 150 lines of code [1] for managing the list of recently used files. With history variables, tasks like this can be accomplished more easily. A history variable, with a depth equal to the number of files to maintain, could be used to store a list of recently used files. Actions such as adding a new file, or retrieving a previously used file from the list are handled by the history variable rather than the programmer. This reduces the amount of code needed, lowers the probability of programming errors, and simplifies debugging.

1.3 History variables as a language feature

The concept of applying history to variables (according to our definition) in a programming language was first proposed by Takaoka, et. al. [70]. Their work focused only on history for scalar variables and remained largely theoretical. A few languages have provided some form of history mechanism as a

[1] Relevant source code available from: `http://www.koders.com/java/fid59D61779E19A7D931AA3DE709AD49B5C9491FD6F.aspx`

feature (we discuss these languages in section 2.3). However, the history support in these languages is typically limited: only allowing history storage to be applied to specific variables types, used in certain situations or restricting the control a programmer has over the history mechanism. To the best of our knowledge, no existing language supports a complete implementation of history variables according to our definition. In this thesis we aim to extend the work of Takaoka, et. al. to develop a full implementation of history variables as a language feature in an imperative programming language and fill the void in this research area.

1.4 Contribution of this thesis

In this thesis we improve and extend the work of Takaoka, et. al. [70]. We provide formalised semantics for the primitive history variables they introduced, and improve their original implementation to allow indirect assignments (i.e. via a pointer) to be made to history variables.

We extended their work on history variables by introducing a new programming language construct called an "atomic block", which allows the history logging of a variable to be temporarily suspended. We also extend the formal semantics and implementation of history variables to other variable types including arrays and structures. Finally we develop an experimental language and compiler which allow us to test the practical performance of our history variable implementations.

1.5 Implementation goals

It is not sufficient for a language to simply provide history variables as a feature, they must also be of practical use. We outline a set of goals for implementing history variables in an imperative language which ensure that they are both useful, and practical in real world applications.

- *Performance*: Assigning and retrieving values from a history variable should not be significantly slower than accessing a normal variable.

Ideally this means that the algorithms used for assignment and retrieval should be achievable in constant time, independent of the depth of history. The memory requirements for a history variable must also be reasonable. If space/time trade-offs do exist then programmers should be provided with a choice between an implementation which has fast access time and high memory consumption, and one which has slow access time, but low memory requirements.

- *Complete implementation*: In the same way that a language which supports pointers allows a programmer to create a pointer to any type, a language which supports history variables should allow programmers to create history variables of any type provided by the language. In this thesis we investigate the application of history variables to scalar types, arrays, pointers, strings and user defined types.

- *Compatibility with non-history variables*: History variables should be as compatible as possible with non-history variables. A non-history variable in an existing program should be able to be converted to a history variable of the same type with minimal refactoring work. The value of a non-history variable should be able to be assigned to a history variable and vice-versa. Functions should be able to accept both history and non-history variables of the correct type as arguments.

- *Correctness*: The semantics for history variables should be well defined and verifiably correct. By formalising the semantics we reduce the potential for ambiguous interpretations and improve the understanding for both language implementors and programmers. We use the system of Hoare logic (see chapter 3) to demonstrate the formal correctness of history variables.

- *Simplicity*: History variables should have simple and easy to learn syntax and semantics. The simplicity of a language feature is difficult to quantify and we do not provide any formal measurement in this thesis. We have, however, made a conscious effort to keep the syntax and semantics of history variables as simple as possible. As an informal

measure of simplicity we say that an experienced programmer should be able to learn the full syntax and semantics for an implementation of history variables in a reasonably short amount of time.

1.6 Types of imperative languages

In this thesis we concern ourselves only with the implementation of history variables in a procedural imperative programming language. Where appropriate we discuss the implications of applying certain aspects of history variables to object orientated imperative languages. However, we leave the implementation details beyond the scope of this thesis. We also briefly discuss existing history support in non-imperative programming languages. We separate imperative languages into two main classes:

- *Statically typed*: Languages which perform typechecking only at compile time and do not store any type information at runtime. In general, problems such as pointer aliasing and array bounds checking cannot be verified at compile time. Because of this, code generated by statically typed languages may perform illegal operations at runtime, causing undefined behaviour or crashes. Languages such as C and C++ are statically typed.

- *Dynamically typed*: Languages which store type information at runtime. Dynamically typed languages are type-safe since operations such as pointer aliasing, array bounds, and type casts can be checked at runtime. Dynamically typed languages may also perform some typechecking at compile time. Languages such as Java and C# are dynamically typed.

Where possible we have attempted to develop solutions that can be applied to both statically and dynamically typed languages. When unified solutions are not possible we explain the different approaches taken for the two classes.

1.7 Thesis overview

We begin in chapter 2 with a discussion of various motivations and applications of history variables and a study of the relevant existing research in those areas. Chapter 3 gives an overview of the mathematical system of Hoare logic, which we use as a basis for proving the correctness of history variables in the following few chapters. In chapter 4 we give a more formal definition of history variables, including a formal notation. We examine the application of history to scalar variables: developing the semantics, proving their correctness and finally providing details for implementation.

In chapter 5 we introduce the concept of an *atomic block*, a language construct which allows the history logging of a variable to be temporarily suspended. Again we outline the semantics and correctness of the new construct and discuss the implementation details. We look at the application of history to arrays in chapter 6, and user defined types in 7. In both chapters we define the semantics, along with their formal correctness, and discuss the implementation details. We show how history arrays and user defined types can be combined with atomic blocks to accomplish some interesting tasks.

In chapter 8 we discuss the development of an experimental compiler which provides a full implementation of history variables and atomic blocks as described in the earlier chapters. In chapter 9 we use our experimental compiler to compare the practical performance of the various algorithms we developed with the theoretical performance presented in the previous chapters. Finally in chapter 10 we outline the conclusions of our work, in particular how well we accomplished the goals we outlined in section 1.5, and discuss the potential for further research in this area.

Chapter II

Background and Related Work

In this chapter we investigate the various motivations for history variables and discuss some of the existing approaches to implementing history storage in computer programs.

2.1 Motivations for history variables

The concept of history pervades many areas of computer science and computing in general. Greenberg and Witten [35] discuss a number of uses for history tracking in computer systems. We outline some of the more interesting applications here.

2.1.1 Undo and redo

Many interactive systems provide the ability to undo [8] changes made by the user, often an arbitrary number of times. The redo operation allows users to reverse the effect of an undo operation, effectively "undoing an undo". A number of models for undo and redo exist, ranging from very simple to complex and powerful [27].

Leeman [48] describes a formal approach to implementing undo and redo operations in a programming language. He identifies two different undo operations called backward undo and forward undo. The history of a program is stored as a collection of states, each with an increasing time value. Backward undo restores the program to an earlier state by changing the current time value to that of the desired previous state, i.e. each of the states that

are being undone are lost. Forward undo advances the current time and makes a copy of a previous state. Forward undo treats the undo operation as an editing command, whereas backward undo treats it as a meta-command. The undo function present in many computer applications typically uses the backward undo model [2].

2.1.2 Recently used lists

The temporal locality of reference principle states that: If an object is used, then there is a high probability that it will be used again in the near future. Many applications support this principle by storing lists of recently used objects and allowing the user to navigate and manipulate these lists [43]. The recently visited websites in a web browser or the recently edited files in a drawing program are examples of this. Often the history for a recently used list can be modified. For example a user may be able to remove specific items from a history list.

2.1.3 Persistent data structures

Driscoll, et. al. [25] define three types of data structures: ephemeral, partially persistent and fully persistent. Ephemeral data structures do not maintain any information about their previous state when the are modified. Partially persistent data structures allow all previous versions of a structure to be retrieved, but only its current version to be modified. Fully persistent data structures allow all versions to be both retrieved and modified.

Overmars [60] provides a systematic approach to transforming an ephemeral data structure to a partially persistent one within the context of searching problems. To reduce the amount of storage space needed to maintain the historical versions of a structure, Overmars proposed storing only the differences between each consecutive element in the history. At set intervals in the history a complete structure is stored. To obtain a given historical structure, the previous complete structure is first obtained and then each of the differences applied in order.

Bagwell [9] describes a fully persistent, immutable linked list data structure called the VList. The VList is particularly useful in the implementation of functional languages which have immutable list types. Operations on a VList are typically faster than operations on standard linked list: addition and deletion can be achieved in O(1) time and accessing the i^{th} element takes O(1) time on average. The VList can be used as the basis for a wide range of data structures including: variable length arrays, deques and hash lists.

2.1.4 Versioning

Versioning is a term used to describe systems which maintain historical or alternate versions of data, typically files. Santry, et. al. [65] describe a filesystem called Elephant (the filesystem which never forgets). They state that as disks become cheaper and larger, there is less reason for users to delete files. In Elephant, destructive operations on files such as modification and deletion cause a new version of the file to be created. Mechanisms are provided to enable users to recall or restore previous versions of files and directories.

Version control systems, such as CVS [74], are often used by software developers to track the changes made to a source code repository during development. Version control systems maintain a full history of each file in a repository, allowing developers to roll back to a previous version. There are a number of situations where it is useful to recover an old version of a source file: Introduction of bugs in later versions, performance of the current version is lower than its predecessor or a decision has been made to abandon a new feature that is being developed.

2.1.5 Temporal databases

A temporal database [71] is an extended form of database management system that allows multiple timestamped values to be associated with a single key. The benefit of this approach over a traditional database is that a temporal database can retain information about the past values of keys.

In order to access the historical information in a temporal database, a temporal query language is used. Grandi, et. al. [34] describe a set of standardised extensions for database query languages to support temporal queries. Three new variables types: history variables, version variables and temporal variables are introduced for accessing historical information from a temporal database. History variables are sets of tuples which define the full history of an object. A version variable is a flat tuple consisting of a number of versions of an object and temporal variables are used to denote a specific time-point in an object's history.

Temporal query languages allow queries to be constructed to perform operations of the form: "Find all instances of an object that have ever matched a given expression". Grandi, et. al. [34] use an example of a database which stores information about peoples jobs. Using a temporal query language they demonstrate simple queries which can extract information about people who changed jobs in a given time period or people who worked in a specific job during the time when a given manager was in charge. Two examples of temporal query languages are HotQuel [33] and HotSQL [34].

2.1.6 Reverse execution

Reverse execution is a technique first proposed by Balzer [10] to improve debugging efficiency by allowing a program to be run, or stepped through in reverse. The Leonardo debugger [21], for example, supports reversed execution of C programs. Cook [18] describes three main approaches to storing the information needed to run a program in reverse:

- *Logging*: A complete trace of the information required to execute a program in reverse is stored during a forward execution of the program.

- *Checkpointing*: The entire execution state of a program is periodically stored during a forward execution. It is possible to reverse to a state between to checkpoints by first returning to the early checkpoint and then executing forwards.

- *Reversed Code*: When compiling a program, a reversed version is also produced. The reversed code allows a programmer to use a standard debugger to run a program in reverse.

The logging and checkpointing approaches are of most interest to us because they generate and store information about the history of a program's state during its execution. The logging approach effectively makes every variable within a program a history variable with an unlimited depth. The major limiting factor of the logging approach is the large amount of memory required to store the history information. Sosic [66] contends that not all history information is meaningful and proposed two methods for reducing the amount of stored history data called spatial and temporal selection. Temporal selection allows history logging to be disabled during specific parts of a program's execution, ignoring the history information created during variable initialisation for example. Spatial selection disables history logging for specific parts of a program's memory. Even with a number of optimisations, Sosic showed that full history logging for reverse execution can generate between 30 and 90 kilobytes of data per million executed instructions.

Agrawal, et. al. [3] make the observation that the fine grained history provided by the logging approach is not always desirable. For example, it is often useful during debugging to be able to step over an entire block of code such as a loop or conditional. Their solution is to log the history of code blocks as though they are simple statements, in effect creating checkpoints at the beginning of each block. Associated with each statement block is a change set which contains a list of all variables and their previous values that are potentially modified within the scope of the block.

2.1.7 Program proofs

History variables have been used in proving the formal correctness of computer programs, particularly parallel programs and those involving co-routines [16, 42, 61]. Clint [17] describes a history variable as an unbounded sequence of values which stores the history of a variable in a program. Clint uses history variables in a purely theoretical sense. History variables are used only

to facilitate the proof of a program, not its execution. Statements which assign or retrieve values of history variables, according to Clint's definition, are ignored during the compilation process.

2.2 Methods of history storage

There are a number of existing methods for storing the historical values of a variable in a programming language. Each method has advantages and disadvantages and is suited to different application areas. The major limitation of all of these methods is that they require the programmer to actively think about storing and retrieving the historical values rather than it being a natural extension of the language.

2.2.1 Temporary variables

The simplest method for maintaining the previous value of a variable is to use a temporary variable. For example, functions for swapping the values of two variables often use a temporary variable for maintaining the value of one of the variables so that it can be recalled after the variable has been overwritten. Temporary variables are typically only useful for maintaining a single historical value.

2.2.2 Arrays and linked lists

Arrays and linked lists are useful for implementing depths of history greater than 1. An array with n elements can be used to store a history of depth $n-1$. The first element in the array holds the current value, the second holds the previous value and the n^{th} element holds the oldest value. Modifying the current value requires each of the historical values to be shifted by one place taking $O(n)$ time. Using arrays for history storage is therefore only recommended for low history depths.

An improvement to using arrays for history storage is to use a cyclic buffer. A cyclic buffer has a pointer to the current value. The historical values follow the current value in chronological order, wrapping around the end of

the array. Cyclic buffers exploit the fact that the oldest value in the history is lost when a new value is added. The new value writes over the oldest value and the pointer is altered to point at the new current value. Assignment and retrieval using cyclic buffers takes O(1) time.

Linked lists (or dynamically allocated arrays) are useful for implementing large, or unbounded depths of history. New values can be added to a linked list in O(1) time. For a finite depth of history the tail node in the list is deallocated when adding a new node to the head if the length of the list is equal to the depth of history. One of the major drawbacks to using linked lists is that the search time is typically $O(n)$.

2.2.3 Function calls

In most modern computer architectures a pointer to the current instruction being executed is stored by the program counter (PC) register. The variables that are local to a function may be stored in local registers or in the current stack frame.

Whenever a function call is made, the computer hardware saves the state of the previous function by storing the program counter and the local variables on the stack, and then allocating a new stack frame for the called function. The called function is then free to write to the program counter, local registers and stack frame. When the function returns, the computer hardware will deallocate its stack frame and restore the program counter and local registers of the calling function. The depth of history for function calls is limited only by the computer's stack space.

Some architectures, notably the Sparc [67], provide more efficient history storage of the local registers during function calls by using register windows [75]. Register windows are additional sets of the local registers. When a function call is made, rather than saving the local registers to the stack, the current register window is simply changed to point at a new set of local registers. If the function call depth exceeds the number of register windows then it becomes necessary to store the registers to the stack. The register windows act as a low depth, efficient history storage of the local registers.

2.2.4 Recursion

Recursive functions offer a limited capability for storing history information. A recursive implementation of a depth first traversal of a tree structure implicitly stores which nodes have been visited and which node to return to when the search of the current subtree is exhausted. An iterative approach requires the programmer to explicitly maintain a list of visited nodes.

Recursion gives a potentially unlimited depth of history, but suffers from the overhead of the function calls and the limited control over the historical data. Recursion implicitly stores history on the function call stack. When a recursive function call (other than the first) exits, the state of its local variables are restored to the values contained by the instance of the function which called it. Once the base case for a recursive function is met and the function is exited the history information is no longer available.

2.2.5 The command pattern

The command pattern [32] is a design pattern used in object orientated programming which encapsulates actions in a program as objects. The command pattern is often used to implement an undo feature in a program. All undoable actions are mapped to objects which have both a method to carry out the action, and a method to undo the changes caused by the action. A stack of action objects is kept. The program can be returned to an earlier state by popping action objects off the stack and executing their undo method.

Use of the command pattern may still require the programmer to manually maintain the history of some values. An example given in [31] uses the command pattern to implement a ceiling fan with an undoable speed setting. The command class used for setting the fan's speed stores the previous speed, in addition to its current speed, so that it can be restored if the undo method is called.

2.2.6 The memento pattern

The memento pattern [32] can be used to store the history of an object's states. The memento pattern works by storing a copy of all the objects that are modified during a specific action. The memento pattern differs from the command pattern by storing information about states rather than actions. Using the command pattern a previous state of a program is recovered by reversing the actions made between the previous and current states, while the memento pattern returns to a previous program state by restoring the individual states of each of the saved objects. The memento pattern is particularly useful if a program may need to recovery from an uncompleted operation. One of the largest drawbacks of the memento pattern is that it can be both slow and memory intensive [31] in certain situations.

2.3 Existing language support

Few existing languages provide support for storing the history of variables. A number of different approaches have been taken to applying history storage in the context of programming languages. We outline some of the more interesting applications of history in programming languages below.

The Interlisp programming environment [45] provides history manipulation commands, such as undo and redo, which can be applied to events in a program. Events are destructive operations such as the modification of a variable. The history mechanism in Interlisp stores a finite list of events, the size of which may be specified by the user. When a new event occurs the information about the oldest event is lost. The state history in Interlisp is stored in a cyclic buffer [72]. The history commands in Interlisp are meta-events. A special syntax is used for performing history commands on events which are history commands themselves. This allows the history mechanism in Interlisp to function as both a forward and backward undo.

In the Icon programming language [36], failure of a conditional expression causes a program's state, prior to the failed expression, to be restored. This is known as data backtracking. To facilitate data backtracking, Icon provides

a reversible assignment operator which maintains a copy of the assigned variable's previous value. If an expression fails the effect of the reversible assignment operator is undone. The reversible assignment operator in Icon only maintains a history of depth 1. Icon does provide any form of manual history manipulation for variables.

The Pikt scripting language [59] provides basic support for single depth history variables. An interesting aspect of the history support in Pikt is that the value a history variable contained in a previous execution of a script can be recalled. Pikt uses different syntax for recalling values from a previous statement and recalling values from a previous execution. In the latter case modifications to a history variable are written to a file which stores the name of the variable, its assigned value, and the line number of the modifying statement.

While each of these languages provides some form of history mechanism, their implementations all differ from our proposed goals (see section 1.5). The Interlisp implementation does not allow history logging to be specified for individual variables, Icon does not allow programmers to manually access the history of a variable and Pikt doesn't support history variables with depths greater than 1.

2.4 Existing library support

Many library frameworks include support for implementing undoable actions in applications using the command pattern. Some supporting frameworks include: The Java Foundation Classes [1] (JFC) [76], the MacApp class library [6] and the .NET framework [2] [54]. Although support for the undo operation using the command pattern is provided by many existing frameworks, it is often overlooked. For example, in a tutorial on design patterns in Java, Cooper [19] describes an implementation of the undo operation in Java by constructing the command pattern from scratch rather than using the existing classes present in the JFC.

[1] As of version 1.1.

[2] As of version 2.0.

The GNU history library [64] can be used for maintaining a history of input lines for programs such as interpretive shells [47]. Functions are provided for adding, retrieving and removing items from the history. The GNU history library has been used in the development of the Python programming language [63] and the MySQL database server [56].

2.5 Summary

There a wide range of motivations for maintaining the history of objects in a computer program: storing a history of actions in a program allows the user to return the previous state if they make an error, recently used lists allow users to quickly recover a document or item they have used in the past and reverse execution can improve the usability of debuggers by allowing users to step backwards from a breakpoint.

The implementation of history storage can be accomplished in a number of different ways, each with different strengths and weaknesses. A few existing languages provide some form of history logging or history variable support. However, the use of history variables in these languages is typically restricted to specific situations or variables types. Existing library support for storing history, especially for undo operations, is abundant with support provided in each of the major application programming frameworks. However, we showed that this support is not always used and programmers occasionally "reinvent the wheel" when implementing history storage.

Chapter III

Hoare Logic

Throughout the next three chapters of this thesis we use the system of Hoare logic [39, 7] to demonstrate the correctness of the various new programming constructs we introduce. This chapter serves as a brief introduction to Hoare logic and the basic axioms that we will be using. For an in-depth treatment of Hoare logic we refer the reader to the book "Design of Well-Structured and Correct Programs" by Alagic and Arbib [5].

3.1 Why do we use Hoare logic?

The system of Hoare logic allows us to reason about the correctness of a program according to a series of predefined axioms. Given a set of preconditions and a program, we can formally show that execution of the program will match a set of expected postconditions, and is therefore formally correct.

We also use Hoare logic as a formalisation mechanism for the informal semantics of the various types of history variables we introduce in this thesis. By introducing new axioms in Hoare logic for history variables and then using those axioms in proving example programs we can demonstrate that our semantics are formally correct. Full or partial axiomatic semantics in Hoare logic have been developed for a number of languages including: Pascal [40], Euclid [49] and Java [58].

3.2 Hoare triples

The central concept of Hoare logic is the Hoare triple. A Hoare triple is of the form:

$$\{P\}S\{Q\}$$

where P and Q are assertions in predicate logic and S is a program or statement. We call P the preconditions and Q the postconditions. The above triple is read as: if P holds before execution of S then Q holds after the execution of S. We express a valid Hoare triple or assertion by prefacing it with the $\models$ symbol. A fuller treatment of validity in Hoare logic is given in [23].

3.3 Definition of axioms

In order to use Hoare triples to prove the correctness of programs we need to define a series of axioms for the basic constructs in a programming language. We have two rules for implication, called the pre-implication and post-implication rules. The pre-implication rule is given as:

$$\frac{P \supset Q\,,\, \{Q\}S\{R\}}{\{P\}S\{R\}} \tag{3.1}$$

The horizontal line is an inference notation. If the top part of the axiom can be derived (i.e. $\models P \supset Q\,,\, \{Q\}S\{R\}$), then by inference, the bottom part is also true. The pre-implication rule states that if an assertion P logically implies Q, and Q is used as the preconditions for a statement S, then P can also be used as the preconditions for S. Similarly, the post-implication rule is given as:

$$\frac{\{P\}S\{Q\}\,,\, Q \supset R}{\{P\}S\{R\}} \tag{3.2}$$

The correctness of a series of statements is proved using the sequential composition axiom:

$$\frac{\{P\}S_1\{Q\}\,,\, \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}} \tag{3.3}$$

In the top part of the sequential composition rule we use an intermediate assertion called Q which serves as both the postconditions for S_1 and the preconditions of S_2, therefore we can discard Q in the bottom of our rule since R will always hold after the sequential execution of S_1 and S_2 with the preconditions P. The axiom for while loops is defined as:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}\texttt{while B do S done}\{P \wedge \neg B\}} \tag{3.4}$$

We call P the loop invariant since it holds true before, during and after execution of the while loop. If the condition B is true before entering the loop, then it must be false at the exit of the loop. Therefore if the body of the loop (top part) is shown to be correct, then it follows that the loop itself (bottom part) must also be correct. Finally we define an axiom for assignment statements:

$$\{P^x_e\}\texttt{x := e}\{P\} \tag{3.5}$$

Here the preconditions are the same as postconditions except that all free occurrences of the variable x are replaced with the expression e.

3.4 Array assignment

Intuitively it may appear that the axiom for standard assignment (3.5) can be directly extended to arrays by substituting the expression e for some array expression $a\texttt{[}i\texttt{]}$ on the left-hand side. In practice, however, this does not always work. Consider the following example:

$$\{a\texttt{[1]} = 1 \wedge a\texttt{[2]} = 3\}\texttt{a[a[1]] := 2}\{a\texttt{[}a\texttt{[1]]} = 2\}$$

This is not a valid Hoare triple. The statement `a[a[1]] := 2` sets the value at $a\texttt{[1]}$ to 2. Our postcondition is therefore false, since $a\texttt{[}a\texttt{[1]]} = 3$ after execution of the assignment. A valid Hoare triple for this assignment is:

$$\{a\texttt{[1]} = 1 \wedge a\texttt{[2]} = 3\}\texttt{a[a[1]] := 2}\{a\texttt{[1]} = 2 \wedge a\texttt{[2]} = 3\} \tag{3.6}$$

However our assignment axiom (3.5) cannot be used to prove this formally, since $a\texttt{[}a\texttt{[1]]}$ does not appear in the postconditions. The solution is to instead map occurrences of an array expression to a function we call α. The array assignment axiom is given as:

$$\{P^{\,a}_{\alpha(a\,,\,i\,,\,e)}\}\texttt{a[i] := e}\{P\} \tag{3.7}$$

with the auxiliary assertion:

$$\alpha(a\,,\,i\,,\,e)\texttt{[}j\texttt{]} \equiv \begin{cases} e & \text{if } i = j \\ a\texttt{[}j\texttt{]} & \text{otherwise} \end{cases} \tag{3.8}$$

Using these two rules we can formally prove the triple from 3.6. We apply the array assignment axiom (3.7) to the postconditions in 3.6. The variable i in the array assignment axiom and auxiliary assertion (3.7 and 3.8) represents the array index $a\texttt{[1]}$. We group each of the cases for i from 0 to $n-1$ using *or* connectives:

$$P^{\,a}_{\alpha(a\,,\,i\,,\,e)} \equiv \bigvee_{i=0}^{n-1} (a\texttt{[1]} = i \wedge \alpha(a\,,\,i\,,\,2)\texttt{[1]} = 2 \wedge \alpha(a\,,\,i\,,\,2)\texttt{[2]} = 3) \tag{3.9}$$

Although the value of $a\texttt{[1]}$ may be outside of the array bounds we do not examine these cases because it would make the statement in 3.6 semantically incorrect and therefore the triple would be invalid. For a treatment of error states in Hoare logic see [73]. Expanding 3.9 and applying the auxiliary assertion (3.8) gives:

$$\begin{aligned} & (a\texttt{[1]} = 0 \wedge a\texttt{[1]} = 2 \wedge a\texttt{[2]} = 3) \\ \vee\ & (a\texttt{[1]} = 1 \wedge 2 = 2 \wedge a\texttt{[2]} = 3) \\ \vee\ & (a\texttt{[1]} = 2 \wedge a\texttt{[1]} = 2 \wedge 2 = 3) \\ \vdots\ & \\ \vee\ & (a\texttt{[1]} = n-1 \wedge a\texttt{[1]} = 2 \wedge a\texttt{[2]} = 3) \end{aligned} \tag{3.10}$$

All of the clauses in 3.10 are false except for the second case ($i = 1$). The second clause is logically implied by the preconditions in 3.6, and we therefore

accept the triple as correct.

3.5 Structure member assignment

Assignment to a member (also called a field) in a structure (also called a record) is a simple extension of the standard assignment axiom (3.5). The axiom for structure assignment, where m is a member of a structure s is:

$$\{P^{s.m}_{e}\}s.m := e\{P\} \tag{3.11}$$

This axiom may also be used for assignment to members of a class.

3.6 Proving correctness

We construct a proof of a program by applying the substitutions in the post-conditions for the assignment statements in reverse order, until the converted postconditions logically implied by the preconditions. If we prove the correctness of each statement within a program, then by the sequential composition axiom (3.3) we infer that the entire program is also correct.

We write the proof of a statement in Hoare logic by first writing its corresponding Hoare triple. We then write the substituted postconditions, and each of our working steps directly below the original preconditions. Formally we write this as:

$$\begin{aligned} &\{P\}S\{Q\} \\ &\{Q_1\} \\ &\vdots \\ &\{Q_r\} \end{aligned} \tag{3.12}$$

where Q_1 is the substituted assertion of Q and $Q_2 ... Q_r$ are our working steps.

If $P \supset Q_r$ then we can say that $\models \{P\}S\{Q\}$. For example:

$$\begin{array}{c} \{x = 1\}\texttt{x := x + 1}\{x = 2\} \\ \{x + 1 = 2\} \\ \{x = 1\} \end{array} \tag{3.13}$$

The assertion on the bottom left-hand side is logically implied by the pre-conditions. Therefore the above Hoare triple is formally correct.

3.7 Weakest preconditions, strongest postconditions

When constructing assertions for statements we want to have the weakest possible precondition and the strongest possible postcondition. An assertion Q is the strongest postcondition if $\models \{P\}S\{Q\}$ and $Q \supset R$ for all R such that $\models \{P\}S\{R\}$. In other words, if we have the strongest postcondition then we can infer all weaker postconditions from it. Similarly an assertion P is the weakest precondition if $\models \{P\}S\{Q\}$ and $R \supset P$ for all R such that $\models \{R\}S\{Q\}$.

3.8 Partial and total correctness

A Hoare triple is said to be totally correct if it is valid, i.e. $\models \{P\}S\{Q\}$ and the termination of S can be guaranteed. When the termination of S cannot be ensured, such as a loop that may run forever or a statement which potentially causes an error, we say that it is partially correct. For simplicity, we give only partially correct proofs. A discussion of total correctness for the axioms given in sections 3.3 – 3.5 can be found in [73].

Chapter IV

Primitive History Variables

A history variable is a variable of a type t with a depth of history d. A primitive history variable is any scalar variable that has an associated depth of history. In this chapter we give a thorough definition of primitive history variables, including the semantics, type information and language syntax. The basic semantics for history variables introduced in this chapter are also used for the other history types we develop later in this thesis.

4.1 Representing history variables

We use the notation $x\langle i\rangle$ to refer to the historical value of the variable x at depth i. The value $x\langle 0\rangle$ is the current value, $x\langle 1\rangle$ is the previous value and $x\langle d\rangle$ is the oldest history of a history variable x of depth d.

We can represent the complete list of values stored by a history variable as an ordered set. For example a history variable x of depth 2 that has a current value of 3 and historical values of $x\langle 1\rangle = 2$ and $x\langle 2\rangle = 1$ can be represented as: $x = \langle 3\,,\,2\,,\,1\rangle$.

The initial values for the current and historical values of a history variable are subject to the implementation language. Throughout this thesis we use the convention that the initial values are undefined, represented by the $\perp$ symbol.

4.1.1 Type information

We use a simple type notation where the type of a variable is given as a concatenation of subtypes separated by the $\rightarrow$ symbol. For example a double indirect integer pointer is represented as: $pointer \rightarrow pointer \rightarrow int$. Subtypes can have some attached information, for example an integer array with 5 elements is expressed as: $array(5) \rightarrow int$.

The type of a primitive history variable for a scalar type t with a depth of d is given as: $history(d) \rightarrow t$. The type of the current and historical values is t. The current value of a history variable is treated as an lvalue, while each of the historical values are treated as rvalues. An lvalue is an object, such as a variable name, which may appear on the left-hand side of an assignment statement. A rvalue is an object, such as an expression or constant value, which may appear only on the right-hand side of an assignment statement. By making the historical values of a history variable rvalues it is not possible to assign to them directly.

4.1.2 Language syntax

We avoid providing a concrete syntax for history variables since the choice of syntax rules, keywords and symbols to be used is largely dependent on the target language for implementation. In chapter 8 we introduce an experimental language, called HistoryC, which supports history variables. A complete BNF grammar for HistoryC is given in appendix A. Throughout this thesis we use a Pascal-like syntax for illustrating code examples. We extend the standard syntax of Pascal to incorporate history variables. A history variable can be declared in the following manner:

```
var int x⟨3⟩;
```

Which declares a history variable x of type $history(3) \rightarrow int$. When assigning to a history variable we need only place the name of the variable on the left-hand side of the assignment statement:

```
x := 10;
```

When a history variable appears on the right-hand side of an expression, we only specify the depth for historical values, not the current value. For example:

```
x := x + x⟨1⟩;
```

Allowing the current value of a history variable to be appear without a depth specifier makes program statements more readable. We also introduce a syntax for declaring pointer variables:

```
var int ↑p;
```

This declares a variable p of type *pointer* $\rightarrow$ *int*. We denote the dereference of a pointer variable as ↑p. Taking the address of a variable x is denoted as ↓x.

4.2 Assignment

History variables are a partially persistent data structure [25], i.e. we can retrieve any historical value, but only modify the current value. Assignment to historical values is prevented by the fact that historical values are rvalues, as described earlier. When we assign an expression e to a history variable x each of the values of x is shifted one history deeper, with the oldest history being lost. This process is shown in 4.1.

$$x\langle d\rangle \leftarrow x\langle d-1\rangle \, , \ldots , x\langle 1\rangle \leftarrow x\langle 0\rangle \, , x\langle 0\rangle \leftarrow e \tag{4.1}$$

It is not possible to copy the entire contents of one history variable to another using a single assignment since this would violate our rule against changing historical values.

4.3 Retrieval

A value is retrieved from a history variable whenever the current value or one of the historical values appears in an expression (except on the left-hand side of an assignment). The depth of history to retrieve from may be specified as

either a constant value or an expression. Retrieval of a value that is undefined or outside the bounds of the history depth can be handled in a number of ways:

- Evaluate as an undefined value. If the accessed value is outside the bounds of the history depth then the behaviour of the program is undefined. This approach should generally be used only for statically typed languages.

- Perform static (compile time) checks to ensure that illegal values are not accessed. History bounds checking is very similar to array bounds checking, and it has long been known that static bounds checking for arrays is, in general, an undecidable problem [69]. This method could be useful for issuing compile time warnings about possible depth bounds violations.

- Perform dynamic (runtime) checks and generate exceptions or error messages for illegal accesses. This method is preferable for dynamically typed languages. Performing static bounds checks where possible can be used to decrease the runtime overhead of the dynamic checks.

4.4 Formal correctness

We extend Hoare's assignment axiom (3.5) to primitive history variables by adding substitutions for each of the historical values to the substitution list in the preconditions. Our axiom for assignment to a history variable of depth d is given as:

$$\{P^{\substack{x\langle 0\rangle,x\langle 1\rangle,\ldots,\ x\langle d\rangle \\ e,\ x\langle 0\rangle,\ldots,x\langle d-1\rangle}}\}\mathsf{x\ :=\ e}\{P\} \tag{4.2}$$

In the left-hand side, the value $x\langle 0\rangle$ is substituted by the expression e. Each other historical value is substituted by the historical value preceding it in chronological order. The value of $x\langle d\rangle$ is lost.

In many ways, history variables are similar to arrays. In section 4.7.1 we will discuss how history variables can be stored as arrays. However, the formal

logic of history variables is significantly different to the formal logic used for arrays (see section 3.4). Because we do not allow the past values of history variables to be assigned to (see section 4.2) it is not necessary to use an auxiliary assertion as it is with arrays (see 3.8). Assignment to an array can only change the value at a single index, whereas assignment to a history variable changes the value at each depth.

4.4.1 Example: Fibonacci series

We illustrate the use our axiom for history assignment (4.2) by proving the formal correctness of a small program used to calculate the first n numbers in the Fibonacci sequence. Mathematically we represent the Fibonacci function as:

$$fib(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } i = 1 \\ fib(i-1) + fib(i-2) & \text{if } i \geq 2 \end{cases} \tag{4.3}$$

The code for a function, which calculates the $n-1^{\text{th}}$ number (counting from zero) in the Fibonacci sequence using a while loop and a history variable of depth 1, is given in listing 4.1.

```
 1  function fib(n) begin
 2    var int i, f⟨1⟩;
 3
 4    i := 2;
 5    f := 0; f := 1;
 6
 7    while i < n do
 8      f := f + f⟨1⟩;
 9      i := i + 1;
10    done
11
12    return f;
13  end
```

Listing 4.1: Function for calculating the $n-1^{\text{th}}$ number in the Fibonacci sequence.

We have provided the code for the full function for completeness. However, we are interested only in proving the correctness of the while loop. The correctness of the statements on lines 4 and 5 of listing 4.1 is clearly provable using axioms 3.5 and 4.2 respectively. We call the statement on line 8 S_1 and the statement on line 9 S_2. The precondition P is defined as:

$$P \equiv f\langle 0\rangle = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i \leq n \tag{4.4}$$

The loop condition is given as: $B \equiv i < n$. We define Hoare triples for S_1 and S_2 as follows:

$$\{P \wedge B\}S_1\{f\langle 0\rangle = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 2 \leq i < n\} \tag{4.5}$$

$$\{f\langle 0\rangle = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 1 \leq i < n\}S_2\{P\} \tag{4.6}$$

We call the intermediate assertion, the postcondition in 4.5 and precondition in 4.6, Q. Using the sequential composition rule (3.3) and expanding P in our postconditions for clarity, we therefore have the following for the top part of the while loop axiom (3.4):

$$\{P \wedge B\}S_1; S_2\{f\langle 0\rangle = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i \leq n\} \tag{4.7}$$

Note that since P contains the assertion $i \leq n$ and B contains the assertion $i < n$. Our preconditions contain only the assertion $i < n$ which satisfies both P and B. We work backwards through the body of the while loop from S_2 to S_1. Applying the assignment axiom (3.5) to the postconditions in S_2 gives:

$$\begin{aligned} &\{Q\}S_2\{P\} \\ \{f\langle 0\rangle = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 2 \leq i+1 \leq n\} \\ \{f\langle 0\rangle = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 1 \leq i < n\} \end{aligned} \tag{4.8}$$

which is logically implied by the preconditions of S_2 (from 4.6). We continue

backwards, applying our history assignment axiom (4.2) to S_1:

$$\frac{}{\{f\langle 0\rangle + f\langle 1\rangle = fib(i) \wedge f\langle 0\rangle = fib(i-1) \wedge 2 \leq i < n\}} \quad \{P \wedge B\} S_1 \{Q\}$$

which can be rearranged as:

$$\{f\langle 0\rangle = fib(i-1) \wedge f\langle 1\rangle = fib(i) - fib(i-1) \wedge 2 \leq i < n\} \tag{4.9}$$

We can rearrange the clause for $f\langle 1\rangle$ in the postconditions according to 4.3 since:

$$fib(i) - fib(i-1) = fib(i-2) \qquad \text{for } i \geq 2$$

which gives:

$$\{f\langle 0\rangle = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i < n\} \tag{4.10}$$

This is logically implied by the precondition $P \wedge B$. Therefore we can conclude that the body of the while loop is formally correct, and, by inference (see 3.4), the entire while loop is also correct. Figure 4.1 shows an inference chart of this proof. Similar charts can be constructed for the other proofs given in this thesis.

4.5 Accessing history variables indirectly

Many imperative languages provide some mechanism for one variable to become an alias of another. In languages like C, this is achieved by providing an explicit pointer type. In modern languages such as Java and C#, all variables except for the scalar types are reference types. In this thesis we refer to both pointers and reference types simply as pointers. Following our completeness goal (see section 1.5) we need to provide a set of semantics for using history variables in the presence of pointers.

Reasoning about pointer programs in Hoare logic remains an open ended problem. A number of different approaches to formally verifying programs

$$\{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) 2 \leq \wedge i < n\} S_1 \{f = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 2 \leq i < n\},$$
$$\{f = fib(i) \wedge f\langle 1\rangle = fib(i-1) \wedge 1 \leq i < n\} S_2 \{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i \leq n\},$$
$$2 \leq i < n \supset 1 \leq i < n$$

$$\{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i < n\} S_1; S_2 \{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) 2 \leq \wedge i \leq n\}$$

$$\{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i < n\} S \{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i \leq n\}$$

$$\{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge 2 \leq i < n\} S_0 \{f = fib(i-1) \wedge f\langle 1\rangle = fib(i-2) \wedge i = n\}$$

Figure 4.1: Proof chart for the history based Fibonacci program in listing 4.1. S_1 and S_2 represent the statements on lines 8 and 9 and S represents the sequential composition of S_1 and S_2. S_0 represents the program statement: `while i < n do` S `done`.

which contain pointers have been considered by various authors [50, 46, 12]. Bornat, et. al. state in a recent paper [13] that the problem of formally reasoning about pointer programs does not yet have a complete solution. We do not introduce formal axioms for reasoning about the correctness of accessing history variables indirectly in this thesis.

4.5.1 Pointers

Pointer variables may only reference the current value of a history variable, not the historical values. This prevents the historical values of a history variable from being modified indirectly. Since the current value of a history variable of type $history(d) \rightarrow t$ is of type t we can reference it with a variable of type $pointer \rightarrow t$.

In languages with explicit pointer types, pointers are assigned to the location of a variable using a unary 'address of' operator. Such languages typically provide a semantic rule which states that operand to the 'address of' operator must be an lvalue. This rule already prevents pointers from referencing the historical values of a history variable since they are rvalues. In a language which does not provide this semantic rule, or does not use an 'address of' operator for pointers, a new semantic rule must be introduced preventing a pointer from referencing a historical value.

We outline two possible semantic rules for indirect assignment to a history variable via a pointer:

1. Indirect assignment to a history variable behaves the same as a direct assignment, updating the history information.

2. Indirect assignment to a history variable changes the current value, but does not update the history.

Consider the program in listing 4.2 which has both direct and indirect assignments to a history variable x. We assume that a standard integer pointer can be used to reference an integer history variable. Following the first semantic rule we have $x = \langle 5\,, 4\,, 3 \rangle$ at the end of the program. If we follow the second

semantic rule we have $x = \langle 5, 4, 1 \rangle$. Note that the value 2 has been lost since the indirect assignments on lines 6 and 7 modify the current value without updating the history, i.e. at the end of line 7 $x = \langle 4\ ,\ 1,\ \bot \rangle$. Ideally we would like to use the first semantic rule since the second rule basically treats history variables as normal variables whenever they are accessed indirectly.

```
1  var int x⟨2⟩, ↑p;
2
3  x := 1; x := 2;
4
5  p := ↓x;
6  ↑p := 3;
7  ↑p := 4;
8
9  x := 5;
```

Listing 4.2: An example program showing indirect access to a history variable x via a pointer p.

The second semantic rule can be accomplished using the existing pointer types in a language. We can only indirectly modify the current value of a history variable under the second rule, and since the type of $x\langle 0 \rangle$ is t, we can therefore reference the current value using a variable of type $\mathit{pointer} \rightarrow t$. We cannot use existing pointer types for the first semantic rule since we need to be able to reference the entire history variable, which is of type $\mathit{history}(d) \rightarrow t$.

4.5.2 History pointer type

One approach which allows us to use the first semantic rule is to introduce a new pointer type specifically for referencing history variables. For this purpose we introduce a new type and a new language keyword both called "hptr". We introduce a semantic rule which states that a history pointer may only be used to reference a history variable of the corresponding type. Because accesses to history variables via a history pointer can trivially be identified at compile time it is possible to update the history of variable when

it is modified indirectly. A history pointer may be declared as:

```
var int hptr p;
```

which declares a variable p of type $hptr \rightarrow int$. The variable p can be used to point to any variable of type $history(d) \rightarrow int$, where d is any depth. Although the introduction of history pointers would allow us to indirectly modify history variables while still correctly updating the history, we have the limitation that history pointers are not compatible with ordinary pointers. We cannot use a single pointer type to reference both a history and a non-history variable even when the primitive types are the same. It may be argued that this is ideal since history and non-history variables are different types and should therefore require distinct pointer types. However, this limits the compatibility between history and non-history variables and therefore imposes on our goal for compatibility (see section 1.5). If a programmer adds a depth of history to an existing non-history variable x then they must also modify any indirect accesses to x. This becomes more difficult (i.e. more work for the programmer) if a pointer used to reference the variable x is also used to reference other non-history variables.

4.5.3 By-reference arguments

When a variable is passed by-value to a function, any changes to it inside the body of the function will be lost when the function returns. By-reference arguments pass the address of a variable (rather than the value) to a function. This allows a function to modify the value of a variable which has been passed as an argument, without the modifications being lost when the function returns. Some languages, notably C, treat pointers and by-reference arguments in the same way, while other languages, such as C#, have separate syntax for pointers and by-reference arguments. We will use the C convention that a by-reference argument for a variable of type t is of type $pointer \rightarrow t$.

Any historical value of a history variable of type $history(d) \rightarrow t$ may be passed by value to a function expecting an argument of type t. Therefore, we may also want to be able to pass the address of a history variable of

type $history(d) \rightarrow t$ to a function expecting a by-reference argument of type $pointer \rightarrow t$. We face a similar problem to the one we encountered in dealing with pointers: We either need to introduce a new type for passing history variables by-reference or we need to pass only the current value by reference and lose the ability to update the history within the function. In the former case we provide a new keyword, called "href", to denote a by-reference history variable argument. A function expecting a single argument of type $pointer \rightarrow history(d) \rightarrow int$, where d is any depth, is declared as follows:

```
function foo(int href a)
```

This approach exhibits the same lack of compatibility between history and non-history variables that we encountered with history pointers. In the case of by-reference arguments the problem is more severe since a programmer would need to implement multiple copies of the same function if references to both history and non-history variables could be expected as arguments.

The second semantic rule is simple to implement. A function declared as:

```
function foo(int ↑a)
```

can accept either the address of a variable of type *int* or the address of the current value of a history variable of type $history(d) \rightarrow int$. In the latter case, the history of the variable will not be updated within the body of the function foo.

4.5.4 Solution

Although the first semantic rule (indirect access to a history variable updates its history) is more ideal, we have shown that it is difficult to put into practice without imposing limitations on a language. We show in section 4.7.3 that if the limitations of history pointers are acceptable for a language then history pointers can be implemented reasonably efficiently.

The second semantic rule (indirect access to a history variable does not update the history) obviously limits the usefulness of history variables when

they are accessed indirectly. However, the second rule uses the same syntax and semantics for both history and non-history variables when they are accessed indirectly. This property increases both the compatibility and simplicity (see section 1.5) of history variables. Therefore the second semantic rule is a superior choice.

Our two semantic rules are not mutually exclusive. A language can follow the second semantic rule and provide history pointers for situations where a programmer wants to be able to update the history of a variable through an indirect assignment. A language should therefore implement either the second semantic alone, or both semantics.

4.6 Strings and pointers

There are many approaches to handling strings in imperative programming languages. Older languages, notably C, use arrays or pointers to represent strings, while more modern languages, such as Visual Basic and Delphi, have a scalar string type. Object orientated languages, such as Java and C#, typically have a string class. For languages which treat strings as a scalar type, our semantics for primitive history variables can be directly extended to include history strings. Consider the following program which we call S:

```
var string s⟨2⟩;

s := "Hello world";
s := "foo";
s := s + "bar";
```

The addition operator is used here to concatenate two strings. This gives the following Hoare triple:

$$\{s = \langle \bot , \bot , \bot \rangle\} S \{s = \langle \text{“foobar”} , \text{“foo”} , \text{“Hello world”} \rangle\} \tag{4.11}$$

which can be formally verified using history assignment axiom (4.2). Using strings as history variables is useful for storing finite history lists such as: websites visited in a browser, names of recently used files in a word processor

or recently entered commands in a interpretive shell. Our semantics and assignment axiom (4.2) for primitive history variables can also be used for pointer variables. Consider the following program which we call T:

```
var int a, b, ↑p, ↑hp⟨2⟩;

p := ↓b;
hp := ↓c; hp := p; hp := ↓a;
```

The variable *hp* is a integer pointer with a history depth of 2. Note that this should not be confused with our concept of a history pointer, which is a pointer to a history variable (see section 4.5.2). We represent the program T with the following Hoare triple:

$$\{hp = \langle \perp , \perp , \perp \rangle\} T \{hp = \langle \downarrow a, \downarrow b, \downarrow c \rangle\} \tag{4.12}$$

Notice that in the postcondition we assert that the value of $hp\langle 1 \rangle$ is $\downarrow b$ rather than p. This is because we are storing the value of p at the time of the assignment, not a reference to p. Modification of the pointer variable p has no effect on the history of *hp*. While we cannot assign to the history values of the variable *hp*, we can dereference them. For example:

```
↑x⟨1⟩ := 10;
```

The above statement modifies the value of the variable pointed to by $hp\langle 1 \rangle$, not the history of *hp*. In a language which supports pointers but has no scalar string type (such as C) we can add a depth of history to a variable of type *pointer* $\rightarrow$ *char* to create string history. We can also use pointer history to maintain references to past versions of aggregate types such as structures. We look at applying history directly to structures in chapter 7.

4.7 Implementation

We discuss the implementation of primitive history variables here in an abstract sense. The concrete details, with regard to implementation in a real compiler, are discussed in chapter 8.

4.7.1 Flat storage

The simplest approach to storing a primitive history variable is to use an array. We call this approach the "flat storage system". The current value is stored as the first element, followed by each of the historical values in chronological order. Retrieval of any value works in exactly the same way as an array access, and is achievable in O(1) time. When assigning a new value, each of the old values need to be shifted one place over. The value at the oldest history position is lost. For a history variable with a depth of d we need to perform d such shifts in addition to the assignment, giving us an assignment time of O(d). The flat storage system requires no additional information to be maintained for history variables. Therefore a primitive history variable with type t and depth d requires sizeof(t) $*$ $(d + 1)$ bytes of storage space.

A compiler can handle the assignment and retrieval of history variables either by emitting code directly or by calling runtime functions. For retrieval of values the former approach may be more efficient since retrieval is relatively simple using the flat storage system. Because the algorithm for assignment is more complex, and therefore requires more code to be generated, implementing it as a runtime function may help reduce the amount duplicated code. We discuss the use of runtime functions for handling assignment and retrieval of history variables in more detail in chapter 8.

4.7.2 Cyclic storage

Takaoka, et. al. [70] describes a cyclic memory structure for storing primitive history variables with a finite depth of history. The cyclic storage structure allows both assignment and retrieval for primitive history variables to be performed in O(1) time. The structure consists of history cycle containing the values of the history variable and a cycle pointer, which we call ρ. When referring to the cycle pointer for a specific history variable x, we use ρ_x. The cycle pointer ρ points to the location of the current value in the cycle. The cycle pointer for a history cycle needs to be initialised before use. For history variables with global scope, the initialisation can be done at compile time

since the address is known in advance (global variables have a fixed address in memory). History variables that are local to a function must be initialised at runtime since the address may be different between function calls.

A total of sizeof(t)$*$(d+1)+sizeof($pointer$) bytes of storage space are required to store a primitive history variable using the cyclic storage system. Each of the historical values are stored in chronological order, wrapping around the boundary of the cycle, so that the first history and the oldest history are on either side of the current value. As a convention we show all history cycles with the first historical value immediately to the right of the current value. Figure 4.2 shows an example layout for a history variable with a depth of 3. Note that the cycle pointer does not necessarily need to be located at a consecutive address to the cycle. As a convention we show the cycle pointer ρ located to the left of the cycle in diagrams.

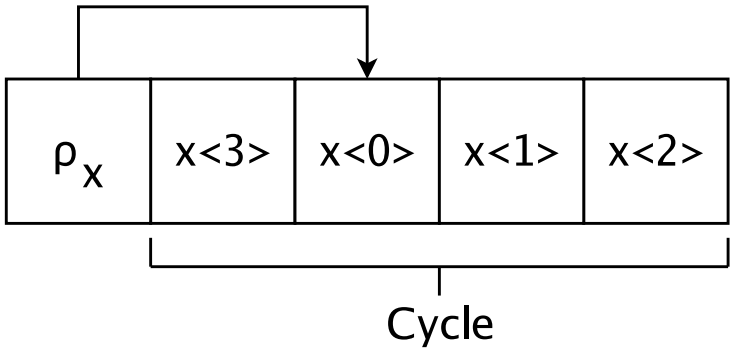


Figure 4.2: Example memory layout using the cyclic storage system for a primitive history variable with a depth of 3.

The cyclic structure takes advantage of the fact that the oldest historical value is lost whenever we assign a new value to a history variable. The new value overwrites the oldest history in the cycle and the cycle pointer is updated to point to the new current value. This process is independent of the history depth and therefore gives us an assignment time of O(1). Retrieval of a given depth i is achieved by first dereferencing the cycle pointer to find the current value, and then counting i places along in the cycle, wrapping around if necessary. The memory address of a value $x\langle i\rangle$ can be found in

O(1) time using the equation given in 4.13. The function $addr(x)$ evaluates the base address in memory of the cycle for the history variable x.

$$pos(x\langle i\rangle) = ((\uparrow \rho_x + (i * \text{sizeof}(t)) - addr(x)) \mod d+1) + addr(x) \quad (4.13)$$

The position equation is complex enough that runtime functions would be more suitable than inline code in most cases for assignment and retrieval.

4.7.3 History pointers

History pointers can be implemented as a pair of fields: an address and a depth specifier. The depth specifier is necessary since a single history pointer may be used to reference many history variables of different depths. Assuming that the depth field is stored as an integer this requires: sizeof(*pointer*) + sizeof(*int*) bytes of storage space. In most modern computer architectures the machine registers are typically sizeof(*pointer*) bytes in size. Unfortunately this means that we cannot store a history pointer in a single machine register, i.e. we require either multiple registers or some main memory storage.

We could store all pointers for a language as history pointers. For example, setting the depth field of a history pointer to zero could indicate that the pointer currently refers to a non-history variable. This would allow a single pointer type to be used interchangeably for both history and non-history variables while providing history updates on indirect assignments to history variables. There are two problems with this approach: It introduces additional runtime overhead for all pointer operations, and code using history pointers to refer to non-history variables will not be compatible with existing code which uses standard pointers. The runtime overhead is caused by necessity of setting or checking the value of the depth field during pointer operations. The compatibility problem is illustrated by Jones and Kelly [44]. They show that an extended pointer type for C, used for bounds checking, is not backward compatible with C's standard pointer type due to the differences in the internal representation. Similarly, history pointers cannot be used as a portable substitute for non-history pointers.

When a history pointer is assigned the address of a history variable, the

address field is set to the location of the history variables cycle pointer (or the base address of the variable if the flat storage system is being used) and the depth specifier is set to the history variables depth. This assignment can clearly be achieved in O(1) time.

Dereferencing a history pointer is handled in the same way as standard assignment and retrieval of history variables, except that a single level of indirection is added. Therefore the theoretical performance for accessing a history variable indirectly is identical to the assignment and retrieval performance of the storage system being used.

4.7.4 The address problem

Cyclic storage performs better than flat storage for assignment and equally for retrieval (at least theoretically, see section 9.4 for a practical comparison) with a minimal overhead in storage space and initialisation time. One major limitation of the cyclic storage system is that the memory addresses of the current and historical values of a history variable are not fixed. Whenever we assign to a history variable that uses cyclic storage the memory addresses of each of its elements are changed. In a language which does not have pointer types this can easily be hidden from the programmer and does not present a problem. Unfortunately this is a very limited subset of imperative programming languages. Even modern languages such as Java and C# (in safe mode) that do not have an explicit pointer type use reference types for most objects. If we change the address of an object in memory we would also need to change the address that any reference variable points to.

We can avoid the address problem by only using history pointers for accessing history variables indirectly. Because history pointers can only be used to reference history variables they can be used to hide the underlying implementation. For example, using the cyclic storage system, dereferencing a history pointer first dereferences the history pointer and then follows the referenced variable's cycle pointer ρ to access the current value. In section 4.5.4 however, we suggested that a language which supports history pointers should also allow non-history pointers to be used for accessing history vari-

ables. Therefore we want to find a solution to the address problem which works for both of the semantic rules we outlined earlier.

The simplest solution for statically typed languages is to use the flat storage system instead of the cyclic storage system. While this removes the address problem completely, it breaks our fast history variables goal (see section 1.5) by increasing the assignment complexity to O(d). Another solution would be to use the flat storage system for history variables that may be referenced by pointers (with a performance penalty) and the cyclic storage system for those that aren't. This becomes a static pointer analysis problem in determining if any other variables "may-alias" [38] a given history variable. Two variables exhibit the may-alias property if they can at any time, in any execution of a program reference the same location in memory. Compile time determination of the may-alias property is found to be NP-Hard even for highly restricted languages [41] and is in general undecidable [37]. Some researchers, such as Steensgaard [68], have investigated algorithms for determining may-aliases that run in close to linear time at the cost of the precision of the results. Although it may be possible to engineer an algorithm specifically for finding may-aliases for history variables that has an acceptable level of precision and a low computational complexity, it is outside the scope of this thesis.

Dynamically typed languages store all type information at runtime. In this case we can solve the address problem by inserting additional runtime checks on indirect assignments. By itself this would incur a large runtime overhead, however by combining runtime checks with some compile time analysis as described above it may be possible to reduce this to an acceptable level. A full analysis of this is, again, outside the scope of this thesis.

4.7.5 Solving the address problem: Extended cyclic storage

We want to provide a general solution to the address problem that can be used for both of our semantic rules, and by both statically and dynamically typesafe languages. The basic problem is that the memory layout for a history variable of type $history(d) \rightarrow t$ using the cyclic storage system is fundamentally different from the memory layout of a variable of type t.

We therefore solve the address problem by making the memory structure of history variables more compatible with normal variables.

Our solution is to move the current value of a history variable outside of the cycle and give it a fixed memory address. Similar to the cycle pointer ρ, the current value of a history variable can now be stored independently of the cycle. As a convention we show the current value located directly to the left of the cycle pointer in diagrams. Figure 4.2 shows an example of the extended cyclic storage layout.

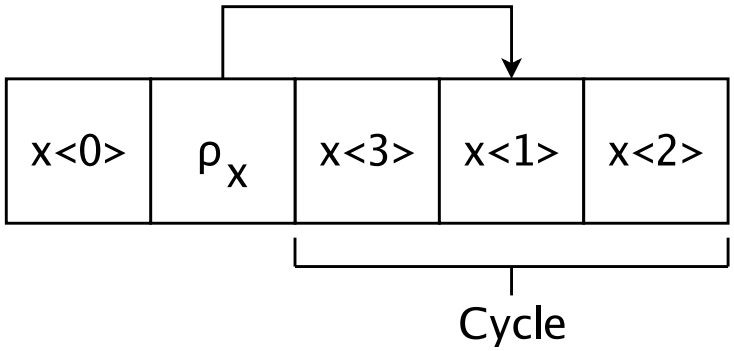


Figure 4.3: Example memory layout using the extended cyclic storage system for a primitive history variable with a depth of 3.

The storage space requirements for the extended cyclic storage system are unchanged from the cyclic storage system. The assignment algorithm for a history variable x is now performed as follows:

- Copy the value of $x\langle 0\rangle$ to the oldest position in the cycle.
- Assign the new value to $x\langle 0\rangle$.
- Update ρ_x to point to the oldest position in the cycle.

This algorithm takes O(1) time. The algorithm for retrieval takes O(1) time. An additional check is required to determine if the value to be retrieved is $x\langle 0\rangle$ or a history value from the cycle. If the depth field of a history variable expression is constant, or otherwise determinable at compile time then these

checks can be made in advance at compile time, otherwise they must be performed at runtime.

Taking the address of a history variable x returns the fixed location of $x\langle 0\rangle$. Indirect assignments therefore modify only the value of $x\langle 0\rangle$ without updating the history, as discussed for the second semantic rule introduced in section 4.5. All indirect modifications to a history variable are lost (not stored in history) except for those immediately preceding a direct modification. For example, consider again the program in listing 4.2. The assignment on line 6 assigns the value 3 to $x\langle 0\rangle$ without updating the history. This is subsequently overwritten with the value 4 by the indirect assignment on line 7. When the direct assignment on line 9 is made the value 7 is copied from $x\langle 0\rangle$ to the oldest position in the cycle, the value 5 is assigned to $x\langle 0\rangle$ and ρ_x is updated. At the end of the program: $x = \langle 5, 4, 1\rangle$. The value of the indirect assignment on line 6 has been lost. However, the value of the indirect assignment on line 7 is stored in the history once the direct assignment on line 9 is made.

4.7.6 Which storage system should we use?

The extended cyclic storage and cyclic storage systems both have theoretical assignment and retrieval times of O(1). Assignment for the extended cyclic storage system requires a total of three assignments to be performed, whereas only two are required for the cyclic storage system. For a language which does not support pointers or reference types, the cyclic storage system may offer slight a improvement in practical performance. However, as discussed earlier, most imperative languages support at least some form of reference type. For languages which do support pointers or reference types, the cyclic storage system cannot be used.

Although the flat storage system has a higher theoretical complexity for assignment than the extended cyclic storage system it may perform better in practice for low history depths. We compare the practical performance of the flat and extended cyclic storage systems in section 9.4. In most situations, we expect that the extended cyclic storage system will provide significantly better practical performance due to the lower assignment complexity, and it is

therefore the best choice for implementation in most imperative programming languages.

4.8 Summary

In this chapter we introduced the concept of primitive history variables and defined a formal notation for their representation. We introduced an axiom in Hoare logic which formalised the semantics we outlined. We demonstrated the use of our axiom in proving the formal correctness of a program which used a primitive history variable to calculate numbers in the Fibonacci series.

We discussed how history variables behave in the presence of pointers and proposed two possible semantics: Indirect access to a history variable has the same semantic rules as direct access, and indirect access to a history variable does not update its history. We introduced the concept of a history pointer for implementing the first semantic. Although the first semantic is more ideal, we showed that the second semantic is simpler to implement and provides better compatibility between history and non-history variables. The implementation of the two semantics is not mutually exclusive and we recommend that a combination of the semantics is the best solution.

We presented three different systems for storing primitive history variables in memory: flat storage, cyclic storage and extended cyclic storage. Flat storage is conceptually simple but hindered by an assignment time of $O(d)$. The cyclic and extended cyclic storage systems both have assignment times of $O(1)$. We introduced the extended cyclic storage system as a solution to the problem that the cyclic storage system fails in the presence of pointers. All three storage systems have a retrieval time of $O(1)$.

Chapter V

Atomic Blocks

Occasionally we may want to "switch off" the history logging for a variable. As discussed in section 2.1.6, Sosic [66] identifies two methods of selective runtime history logging called spatial and temporal selection. Spatial selection allows history logging to be applied only to certain parts of a programs memory. Our approach to history variables already allows spatial selection by relying on the programmer to specify which variables have associated history depths. Temporal selection allows history logging to be suspended during specific parts of a programs execution. There are a number of times when temporal selection is useful:

- Initialising data structures, such as large arrays, can produce a large amount of history information. Usually we are uninterested in the values contained by a data structure prior to its initialisation, and therefore do not want to store the uninitialised values to its history.

- Sometimes a programmer may want to split the calculation for the value of a variable into several assignment statements, but only store the final value into the variables history. The reasons for splitting a single assignment into multiple ones may be purely aesthetic, or the logic of a particular calculation may simply make it difficult to implement it as a single line. Figure 5.1 shows how two otherwise functionally equivalent programs can result in different values being stored to a variables history.

- Although the cyclic storage systems we presented in chapter 4 have a theoretical assignment complexity of O(1) we show in chapter 9 that

```
var int x⟨2⟩;

x := a + b + c;
```

Listing 5.1: Grouped statement.

```
var int x⟨2⟩;

x := a;
x := x + b;
x := x + c;
```

Listing 5.2: Separate statements.

Figure 5.1: Two listings showing equivalent programs that produce different history results for the variable x. At the end of the program in listing 5.1 $x = \langle a + b + c, \perp, \perp \rangle$, while listing 5.2 results in $x = \langle a + b + c, a + b, a \rangle$.

> the practical performance of history variables can be significantly worse than the performance of non-history variables. A programmer may therefore want to suspend the history logging of some variables in parts of a program, such as bottlenecks, where performance is paramount.

We allow programmers to apply temporal selection to history variables by introducing the a language construct called an "atomic block". An atomic block temporarily suspends the history logging of all variables that are assigned to within its body. At the end of an atomic block the history is updated for all history variables that were modified inside its body. We use the name "atomic block" since they cause several assignments to a history variable to be "atomically" added to the history information, resulting in only the last change in an atomic block being saved.

5.1 Representing atomic blocks

Atomic blocks are a statement, and therefore have similar grammatical rules to other block structures such as "while" loops and "if"statements. We use the following syntax for writing an atomic block:

```
atomic begin
  <statements>
end
```

Although atomic blocks are used to suspend the logging for history variables, we allow any statement to be placed inside the body. Only statements which assign a value to a history variable are affected by atomic blocks. Atomic blocks themselves may appear as a statement inside another atomic block. We discuss nested atomic blocks in more detail in section 5.3.

5.1.1 Bound atomic blocks

Sometimes we may want to suspend history updates for only a small number of history variables. We allow atomic blocks to suspend updates for a subset of the history variables in a program by specifying a list of variable names at the beginning of the atomic block. For example:

atomic $(x_1, \ldots, x_n)$ begin
  <*statements*>
end

We say that the above atomic block is explicitly bound to the variables $x_1, \ldots, x_n$, i.e. only the variables $x_1, \ldots, x_n$ have their history updates suspended. Assignments to any other history variables within the body of the atomic block update the history as normal. If an atomic block A binds the history variable x, then x is also bound to any atomic blocks nested within the body of the A. We call this implicit binding. Binding does not extend over function calls. For example if an atomic block binds a variable x, and a function foo is called within the body of the atomic block, then x is not bound inside foo. We discuss this further in section 5.2.3. History updates for x, however, are only performed at the end of an atomic block which explicitly binds x.

5.2 Formal correctness

In order to create a simple axiom for atomic blocks we deal only with atomic blocks which are bound to a single variable x. By extension, the axioms we develop could be used to prove the correctness of atomic blocks that are

bound to many variables, and also unbound atomic blocks (i.e. binds all variables).

Inside the body of the atomic block the history logging of the variable x is suspended. At the end of the atomic block, $x\langle 0\rangle$ will contain the value of the last assignment to x within the atomic block's body. The values of $x\langle 1\rangle ... x\langle d\rangle$ will be equal to the values of $x\langle 0\rangle ... x\langle d\rangle$ immediately prior to entering the atomic block.

We need some mechanism to allow us to assign to the variable x without updating its history inside an atomic block while still maintaining all of necessary information to correctly update the history once the atomic block is exited. We introduce a ghost variable [1], called φ_x, for saving information about a history variable x inside an atomic block. There are two ways that we can use φ_x to save the necessary information. The first method is to save the current value of x in φ_x when entering an atomic block. New values are assigned directly to $x\langle 0\rangle$ (without updating the history) inside the atomic block. At the end of the atomic block the history is updated with the value of $x\langle 1\rangle$ being restored from φ_x. The alternative approach is to suspend writing to x during an atomic block and instead write new values to φ_x. We will use the latter approach because it more closely resembles our implementation.

We start by modifying our axiom for history variable assignment (4.2) so that we assign to φ_x when inside an atomic block that binds x:

$$\{P_0\}\mathsf{x := e}\{P\} \tag{5.1}$$

We define P_0 by an auxiliary assertion:

$$P_0 \equiv \begin{cases} P^{\varphi_x}_{e'} & \text{if } x \text{ is currently bound to an atomic block} \\ P^{x\langle 0\rangle, x\langle 1\rangle, \ldots, x\langle d\rangle}_{e,\ x\langle 0\rangle, \ldots, x\langle d-1\rangle} & \text{otherwise} \end{cases} \tag{5.2}$$

e' is derived from e, where $x\langle 0\rangle$ is replaced by φ_x

[1] As an interesting aside, some authors [17, 42], call ghost variables "history variables" since they are used to preserve values during proofs. Ghost variables have also been referred to as mythical variables [16] and auxiliary variables [62].

Our atomic block axiom is then defined as:

$$\frac{\{P \wedge \varphi_x = x\langle 0\rangle\} S \{Q\}\,,\, Q \supset R_0}{\{P\}\texttt{atomic(x) begin S end}\{R\}} \tag{5.3}$$

where R_0 is defined by an auxiliary assertion:

$$R_0 \equiv R^{x\langle 0\rangle, x\langle 1\rangle ..., \;\; x\langle d\rangle}_{\varphi_x,\; x\langle 0\rangle ..., x\langle d-1\rangle} \tag{5.4}$$

In the preconditions P of the atomic block axiom (5.3) we assert that $\varphi_x = x\langle 0\rangle$. This assertion ensures that the value of φ_x contains the correct value in the expression e' (from 5.2).

5.2.1 A simple example

We demonstrate the use of our atomic block axiom (5.3) using the example program given in listing 5.3.

```
1 var int x⟨2⟩;
2
3 x := 1; x := 2; x := 3;
4
5 atomic(x) begin
6   x := x + 1;
7   x := x + x⟨1⟩;
8   x := x + 2;
9 end
```

Listing 5.3: Simple atomic block example.

We call the atomic block (lines 5 – 9) in listing 5.3 A and the statements within the atomic block (lines 6 – 8) S. We represent our preconditions and expected postconditions as a Hoare triple:

$$\{x = \langle 3\,,\, 2\,,\, 1\rangle\} A \{x = \langle 8\,,\, 3\,,\, 2\rangle\} \tag{5.5}$$

The ghost variable φ_x only exists within the context of an atomic block which binds x, and therefore is omitted from assertions which lie outside the body

of the atomic block. For the assignments in S we expect that $\varphi_x = 4$ at the end of line 6, $\varphi_x = 6$ at the end of line 7 and $\varphi_x = 8$ at the end of line 8. We work backwards through the statements in S applying our new assignment axiom (5.1) to x:

$$\begin{array}{l} \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 6\}\texttt{x := x + 2}\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 8\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x + 2 = 8\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 6\} \end{array} \tag{5.6}$$

$$\begin{array}{l} \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 4\}\texttt{x := x + x}\langle 1\rangle\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 6\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x + x\langle 1\rangle = 6\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 4\} \end{array} \tag{5.7}$$

$$\begin{array}{l} \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 3\}\texttt{x := x + 1}\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 4\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x + 1 = 4\} \\ \{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 3\} \end{array} \tag{5.8}$$

According to auxiliary assignment assertion (5.2), appearances of $x\langle 0\rangle$ have been replaced by φ_x in the substitutions. According to the sequential composition rule (3.3) we now have:

$$\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 3\}S\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 8\} \tag{5.9}$$

From our postconditions in 5.5 we expect that:

$$R \equiv x = \langle 8\,,\,3\,,\,2\rangle \tag{5.10}$$

Applying the auxiliary atomic block assertion (5.4) to R gives:

$$R_0 \equiv \varphi_x = 8 \wedge x\langle 0\rangle = 3 \wedge x\langle 1\rangle = 2 \tag{5.11}$$

We call the postconditions from 5.9 Q. We then have $Q \equiv R_0 \wedge x\langle 2\rangle = 1$ and consequently $Q \supset R_0$. Therefore, according to our atomic block axiom (5.3),

we can conclude that A is formally correct given the pre and postconditions in 5.5.

5.2.2 Empty atomic blocks

According to our atomic block axiom 5.3, if an atomic block binds a history variable x and does not assign to x within its body, then the current value of x is added to the history of x at the end of the atomic block. Consider the following atomic block, which we call A:

```
atomic(x) begin ; end
```

where x is a primitive history variable of depth d. If $x = \langle v_0 , v_1 , ... , v_d \rangle$ at the prior to A, then we have:

$$\{x = \langle v_0 , v_1 , ... , v_d \rangle\} A \{x = \langle v_0 , v_0 , v_1 , ... , v_{d-1} \rangle\} \tag{5.12}$$

Therefore the effect of A is functionally equivalent to:

```
x := x;
```

We use this semantic formally for two reasons: It simplifies the atomic block axiom (5.3) and we do not expect many programs to contain atomic blocks which do not assign to their bound variables. In our practical implementation of atomic blocks (see sections 5.4 and 8.4) we change this semantic so that if a history variable x is not assigned to within an atomic block then its value is unchanged, i.e.:

$$\{x = \langle v_0 , v_1 , ... , v_d \rangle\} A \{x = \langle v_0 , v_1 , ... , v_d \rangle\} \tag{5.13}$$

We use the altered semantic in our practical implementation because it is more intuitive and allows unbounded atomic blocks (i.e. binding all history variables) to be used more effectively.

5.2.3 Binding history variables with global scope

Our atomic block axiom (5.3) cannot be used to verify the correctness of a program where a function, which modifies a global history variable x, is

called from an atomic block which binds x. Consider the following function where x is an integer history variable of depth d with global scope:

```
function foo() begin
  x := x + 1;
end
```

If we call the above function F, then we can represent the general case of a call to foo as:

$$\{x = \langle v_0 \, , ...v_d \rangle\} F \{x = \langle v_0 + 1 \, , v_0 \, , \ldots \, , v_{d-1} \rangle\} \tag{5.14}$$

Atomic block bindings do not extend to over function calls (see section 5.1.1). Therefore, if foo is called from the context of an atomic block which binds x, we may expect:

$$\{x = \langle v_0 \, , ...v_d \rangle \wedge \varphi_x = u\} F \{x = \langle u + 1 \, , v_0 \, , ...v_{d-1} \rangle \wedge \varphi_x = u + 1\} \tag{5.15}$$

where the value of $x\langle 0 \rangle$ has been retrieved from φ_x within foo, and both the history of x, and the value of φ_x have been updated. However, because x is not bound inside foo we follow the second part of our auxiliary assignment rule 5.2. We therefore cannot prove the formal correctness of the Hoare triple in 5.15.

From an implementation point of view, solving this problem would require two separate versions of the function foo. One which reads the current value of x from $x\langle 0 \rangle$, i.e. when foo is not called from the context of an atomic block which binds x, and another which reads from φ_x (or its equivalent in a compiler, see section 8.3.1), i.e. when foo is called from the context of an atomic block. Unfortunately, determining the context from which a function was called, or whether a given function modifies a global variable are both undecidable problems. Therefore we introduce a semantic rule stating that if a history variable x which has global scope is bound to an atomic block then any modification of x by a function called from within the body of the binding atomic block will result in undefined behaviour. This problem does not occur if a bound variable, with either global or local scope, is passed to a function as an argument. A more robust solution to this problem is outside

the scope of this thesis.

5.3 Nested atomic blocks

Like other block structures in a programming language, atomic blocks may be nested. We refer to atomic blocks which are nested inside another atomic block as "inner" blocks and atomic blocks which have other atomic blocks nested within them as "outer" blocks. We make a restriction that an inner atomic block may not explicitly bind any variables that are also bound by an outer atomic block. Consider the example program in listing 5.4 which contains nested atomic blocks.

```
 1  var int x⟨2⟩;
 2
 3  x := 1; x := 2; x := 3;
 4  y := 1; y := 2; y := 3;
 5
 6  atomic(x) begin
 7    x := 4;
 8    atomic(y) begin
 9      y := 4;
10      y := 5;
11      x := 5;
12    end
13    x := 6;
14    y := 6;
15  end
```

Listing 5.4: Example program containing nested atomic blocks.

We call the outer atomic block (lines 6 – 15) A_1 and the inner atomic block (lines 8 – 12) A_2. For the inner atomic block A_2 we expect:

$$\left\{\begin{matrix} x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 4 \wedge \\ y = \langle 3\,,2\,,1\rangle \end{matrix}\right\} A_2 \left\{\begin{matrix} x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 5 \wedge \\ y = \langle 5\,,3\,,2\rangle \end{matrix}\right\} \tag{5.16}$$

The history of x is suspended within A_2, but not updated at the end of it since it is not explicitly bound to the inner atomic block. For the outer atomic block we expect:

$$\{x = \langle 3\,,\,2\,,\,1\rangle \wedge y = \langle 3\,,\,2\,,\,1\rangle\} A_1 \{x = \langle 6\,,\,3\,,\,2\rangle \wedge y = \langle 6\,,\,5\,,\,3\rangle\} \tag{5.17}$$

For clarity we show only the relevant parts of the pre and postconditions at each step. We start by proving the correctness of the inner atomic block A_2, working backwards through the statements using our new assignment rule (5.1):

$$\begin{aligned} &\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 4\}\texttt{x := 5}\{x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 5\} \\ &\{x = \langle 3\,,\,2\,,\,1\rangle \wedge 5 = 5\} \end{aligned} \tag{5.18}$$

$$\begin{aligned} &\{y = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_y = 4\}\texttt{y := 5}\{y = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_y = 5\} \\ &\{y = \langle 3\,,\,2\,,\,1\rangle \wedge 5 = 5\} \end{aligned} \tag{5.19}$$

$$\begin{aligned} &\{y = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_y = 3\}\texttt{y := 4}\{y = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_y = 4\} \\ &\{y = \langle 3\,,\,2\,,\,1\rangle \wedge 4 = 4\} \end{aligned} \tag{5.20}$$

In 5.18 no history update is performed for x since it is implicitly bound to the inner atomic block according to our scope rules (see section 5.1.1). Applying the auxiliary atomic block assertion (5.4) to the postconditions of the inner atomic block (from 5.16) gives:

$$R'_0 \equiv x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 5 \wedge y\langle 0\rangle = 3 \wedge y\langle 1\rangle = 2 \wedge \varphi_y = 5 \tag{5.21}$$

Combining the assertions for x and y from 5.18 and 5.19 for the postcondition of the final statement in A_2 (line 11) gives:

$$Q' \equiv x = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_x = 5 \wedge y = \langle 3\,,\,2\,,\,1\rangle \wedge \varphi_y = 5 \tag{5.22}$$

Clearly Q' logically implies R'_0. Therefore, according to the atomic block axiom (5.3), we accept the inner atomic block as formally correct.

We now prove the correctness of the outer atomic block A_1. The statement on line 14 assigns to the variable y, which is not bound by the outer atomic block. According to our new assignment axiom (5.1) the history of y is therefore updated normally:

$$\begin{array}{l} \{y = \langle 5\,,3\,,2\rangle\}\mathsf{y\ :=\ 6}\{y = \langle 6\,,5\,,3\rangle\} \\ \{6 = 6 \wedge 5 = 5 \wedge 3 = 3\} \end{array} \tag{5.23}$$

The statement on line 13 assigns to the variable x which is bound to the outer atomic block:

$$\begin{array}{l} \{x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 5\}\mathsf{x\ :=\ 6}\{x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 6\} \\ \{x = \langle 3\,,2\,,1\rangle \wedge 6 = 6\} \end{array} \tag{5.24}$$

The next statement in A_1 is the inner atomic block A_2. We have already shown that A_2 is formally correct under the pre and postconditions given in 5.16. The proof of the final remaining statement in A_1 is given as:

$$\begin{array}{l} \{x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 3\}\mathsf{x\ :=\ 4}\{x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 4\} \\ \{x = \langle 3\,,2\,,1\rangle \wedge 4 = 4\} \end{array} \tag{5.25}$$

Each statement in A_1 is therefore formally correct. Applying the auxiliary atomic assertion (5.4) to the postconditions in A_1 (from 5.17) gives:

$$R_0 \equiv x\langle 0\rangle = 3 \wedge x\langle 1\rangle = 2 \wedge \varphi_x = 6 \wedge y = \langle 6\,,5\,,3\rangle \tag{5.26}$$

Combining the assertions for x and y from 5.23 and 5.24 for the postconditions of the final statement in A_1 (line 14) gives:

$$Q \equiv x = \langle 3\,,2\,,1\rangle \wedge \varphi_x = 6 \wedge y = \langle 6\,,5\,,3\rangle \tag{5.27}$$

which logically implies R_0. Therefore we can conclude the formal correctness of the nested atomic blocks in listing 5.4 is correct according to our atomic

block axiom (5.3), given the pre and postconditions in 5.17.

5.4 Implementation

To simplify our implementation, we do not allow nested atomic blocks to share a common explicitly bound history variable. We can determine at compile time if a nested atomic block shares a common explicitly bound variable with an outer atomic block which is in the same function. In this case the compiler should emit an error message.

We describe the implementation at an abstract level here. We discuss the concrete implementation of atomic blocks in our experimental compiler in chapter 8. The compiler maintains a counter, called k, which stores the current nesting level of atomic blocks. We call the atomic block that k refers to A_k. At the beginning of a program k is initialised to 0 (i.e not in an atomic block). For each k we have a set β_k which contains the list of variables which are explicitly bound to A_k, and a set μ_k which stores the list of variables that have been modified within the body of A_k.

When an atomic block is entered the compiler increments the value of k and initialises the modified set to the empty list, i.e.: $\mu_k = \emptyset$. Whenever the compiler encounters an assignment to a history variable x, it is added to the set μ_i where:

$$0 < i \leq k \wedge x \notin \mu_i \wedge x \in \beta_i \tag{5.28}$$

Because we do not allow nested atomic blocks to share common explicitly bound variables, x can only be added to one set μ_i. At compile time it takes $O(\sum_{i=1}^{k} |\mu_i|)$ time, where $|\mu_i|$ is the number of variables in μ_i, to test whether an assignment to a history variable x is currently bound to an atomic block.

When an atomic block is exited, the compiler generates code to update the history of all variables in μ_k and then decrements the value of k. At runtime it takes $O(|\mu_k|)$ time (assuming that the update of a history variable can be performed in $O(1)$ time) to exit an atomic block. The history for a variable x can be updated by assigning the value of φ_x (or its equivalent in a compiler, see section 8.3.1) to x.

5.5 Summary

In this chapter we introduced the concept of an atomic block. Atomic blocks add a mechanism for temporal selection to history variables. We described two types of atomic blocks: unbound and bound. Unbound atomic blocks suspend history updates for all history variables in a program, whereas unbound atomic blocks only suspend the updates for a specified set of history variables.

We formalised our semantics for atomics by developing a series of axioms in Hoare logic. We demonstrated the use of our axioms for atomic blocks with two example programs, one of which contained nested atomic blocks. We discovered that we cannot reason about the formal correctness of a function which modifies a history variable with global scope if the function is called from an atomic block which binds the history variable. We noted that this problem would also exist in a practical implementation, and that determining occurrences of it is undecidable. We therefore introduced a semantic rule stating that the behaviour of a program in such situations is undefined.

Finally we described a system for implementing atomic blocks in a compiler. The runtime complexity of exiting an atomic block is $O(|\mu_k|)$, where $|\mu_k|$ is the number of variables that were explicitly bound to an atomic block A_k and modified within its body. In this chapter we have limited our proofs to trivial examples which demonstrate the semantics of atomic blocks. In the next two chapters we will show how atomic blocks can be combined with other history types to accomplish some more interesting tasks.

Chapter VI

History Arrays

A history array is any array variable that has an associated depth of history. The simplest system for implementing history arrays is to simply store an array of primitive history variables as shown in figure 6.1.

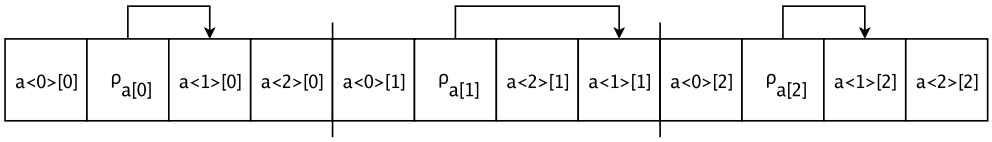


Figure 6.1: Example memory layout using the extended cyclic storage system for a history array with 3 elements and a depth of 2.

This approach stores the history individually for each index in the array, i.e. updating the history for one index in an array does not affect the history of any other index. We call this "index-wise" storage. By using the cyclic or extended cyclic storage system for each index we can perform both assignment and retrieval in O(1) time. An alternative approach to storing the history of an array is store the history of every index (i.e. the entire array) whenever a value is assigned to any index. We call this second method "array-wise" history.

Index-wise and array-wise history are useful in different situations. Index-wise arrays store the history of each index individually, but do not provide any information about relationships between the histories of elements. Array-wise arrays do the opposite: The relationship between elements at each historical depth is maintained, but histories are not stored for individual elements.

For example, an array storing the current salary for each employee at a company could be stored as an index-wise array. The history of changes to each employees salary is maintained individually. Using index-wise history it is possible to retrieve the n^{th} previous salary for any employee. However, given a historical value for one employees salary, it is not possible to determine what any other employees salary was at the same time.

Array-wise arrays are useful, for example, if we want to be able to undo changes to the state of a set of objects which are stored as an array. For example, an array of object positions for a drawing program could be stored using an array-wise array. When an object is moved, the previous state of the array is stored to the history. We look at a more concrete version of this example in section 7.3 when we discuss the application of history to structured types. Array-wise history is also useful in applying history to a fixed-length string. Changes to the current string cause an entire copy of the previous string to be stored in history.

6.1 Representing array history

We use the convention that an array with n elements is indexed as $a\texttt{[}0\texttt{]}...a\texttt{[}n-1\texttt{]}$. Formally we represent the elements of an array a with n elements as: $a = [v_1 , ... , v_n]$. Combining this with our formal representation for history variables introduced in section 4.1 allows us to formally represent the values for a history array. For example an array a with a depth of 1, current values of $a\texttt{[}0\texttt{]} = 1$, $a\texttt{[}1\texttt{]} = 2$ and $a\texttt{[}2\texttt{]} = 3$ and an uninitialised history is represented as: $a = \langle [1 , 2 , 3] , [\perp , \perp , \perp] \rangle$. If we want to refer only to the elements of a at depth 0 we can write: $a\langle 0 \rangle = [1 , 2 , 3]$. We use this notational system for both array-wise and index-wise arrays. When reasoning about history arrays in a formal sense, we always place the depth specifier before the index specifier. For example the element at depth 1 and index 2 of an array a, where a has either index-wise or array-wise history, is represented as: $a\langle 1 \rangle\texttt{[}2\texttt{]}$.

6.2 Multidimensional arrays

In an abstract sense we can visualise a two-dimensional array as a grid. We therefore intuitively think of four different ways to store the history: In addition to index-wise and array-wise we could also store the history of the row or the column that the assigned element is in. While the idea of row and column history storage for a two-dimensional array appears interesting it does not relate well to the actual storage of arrays. A multidimensional array is really just an array of arrays, typically stored in memory using either row-major or column-major order [4]. Consider a two-dimensional array a with n rows and m columns. In row-major format a is stored as an array of n elements, each of which is an array of m elements. In general, for an array with n dimensions, there are a total of 2^n possible methods for storing the history. For a two-dimensional array stored in row-major format, the following approaches for storing history are possible:

- Index-wise for each row, index-wise for the entire array.
- Array-wise for each row, index-wise for the entire array.
- Index-wise for each row, array-wise for the entire array.
- Array-wise for each row, array-wise for the entire array.

Each of these methods results in redundant storage of history information. In the first case for example, assigning to the element at row i, column j in an array a causes the history of the index $a[i\,,\,j]$ to be stored (index-wise for each row) and the history of the entire row $a[i]$ to also be stored (index-wise for the entire array). The other methods above all exhibit some similar form of redundancy. The problem of redundant storage also occurs if we attempt to combine other history types. This is discussed further in section 7.4.

This redundant storage is of little use and supporting many different forms of history storage for multidimensional arrays serves only to complicate the syntax and semantics of a language. We therefore limit multidimensional arrays to only index-wise and array-wise storage over the entire array. Because

multidimensional arrays do not differ from single dimensional arrays in terms of history storage, we will focus the remainder of this chapter solely on single dimensional arrays.

6.3 Index-wise arrays

Index-wise arrays maintain the history of each of their elements individually. We use the following syntax for the declaration of an index-wise array of type t with n elements and a depth of history of d:

$$\texttt{var}\ \ t\ \ \texttt{a}[n]\langle d\rangle\,;$$

We place the depth specifier after the dimension specification in the declaration to signify that the history belongs to each index. The type of the array a from the above declaration is given as: $array(\mathrm{n}) \rightarrow history(\mathrm{d}) \rightarrow t$.

When using index-wise arrays in a formal sense we place the depth specifier before the index specifier. This enables us to simplify our axioms for formal correctness. In the implementation of our experimental language we also use the syntax given above for appearances of index-wise arrays in expressions. Using our declaration syntax allows index-wise arrays to be easily distinguished from array-wise arrays (see section 6.4). However, the syntax used for appearances of index-wise arrays in expressions is subject to implementation.

6.3.1 Contiguous arrays

Some languages specify that arrays must be stored contiguously in memory and allow programmers to exploit this fact. For example the C function memcpy can be used to quickly copy the elements of one array into another. The memcpy function relies on the fact the elements of an array are always stored in the same order.

When implementing history variables in a language which supports this sort of low-level array access we define a semantic rule that states: the current values of each index of a history array must be stored contiguously. This

satisfies our compatibility goal (see section 1.5), allowing a history array to be used in place of an ordinary array without changing the semantics. The example implementation shown in figure 6.1 does not satisfy this rule. Although the memory locations for the current value of each index are fixed, they are not contiguous. Recall that in section 4.7.5 we stated that the cycle pointer, and the current value of a history variable can be stored independently of the history cycle. Therefore we can make the values of the current array contiguous simply by moving and grouping them together as shown in figure 6.2.

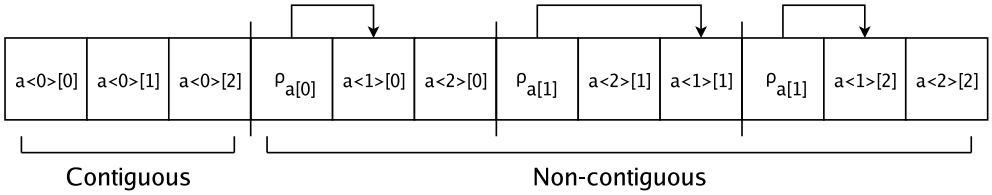


Figure 6.2: Rearrangement of the memory layout from figure 6.1 so that the current values of each index are stored contiguously.

Because we are introducing history variables as a new concept in a language we do not require that the historical values of an array necessarily be stored contiguously. This does not break compatibility with existing code and gives us more flexibility in developing storage systems. If we did require that the historical values of an array also be stored contiguously then we reduce the optimal assignment time for an index-wise array to $O(d)$. For example given an index-wise array a with a history depth of d, we can regard each depth of history as a separate contiguous array. When we assign a value v to an element i in a we need to update the value at index i, in each of the historical arrays as follows:

$$a\langle d\rangle[i] \leftarrow a\langle d-1\rangle[i], \ldots, a\langle 1\rangle[i] \leftarrow a\langle 0\rangle[i], a\langle 0\rangle[i] \leftarrow v$$

In order for the each of the arrays to remain contiguous we cannot change the memory locations of any of the elements. Therefore we require at least $d+1$ assignments to be made, giving us an assignment complexity of $O(d)$.

For languages which hide the underlying implementation of arrays, such as Java and C#, it is not necessary to specify that history arrays must be stored contiguously. In this case the memory layout for an index-wise array shown in figure 6.1 can be used to achieve O(1) assignment and retrieval times. In this chapter we focus on memory layouts for history arrays that have at least the current values stored contiguously (see figure 6.2).

6.3.2 Formal correctness

We develop an axiom in Hoare logic for assignment to index-wise arrays by extending the standard axiom for array assignment (3.5). The axiom is divided into several parts since we need to incorporate the alpha function (as discussed in section 3.4) and the axiom must work in the presence of atomic blocks. We define the array-wise assignment axiom as:

$$\{P_0\}\texttt{a[i] := e}\{P\} \tag{6.1}$$

We define P_0 by an auxiliary assertion which assigns to a ghost variable if a is currently bound to an atomic block (see section 5.2). For both index-wise and array-wise arrays we have an n element ghost array φ_a indexed $\varphi_a\texttt{[0]}...\varphi_a\texttt{[}n-1\texttt{]}$, where n is the number of elements in the array a. The auxiliary assertion for P_0 is given as:

$$P_0 \equiv \begin{cases} P^{\varphi_a}_{\alpha(\varphi_a\,,\,i\,,\,e')} & \text{if } a \text{ is currently bound to an atomic block} \\ P_1 & \text{otherwise} \end{cases} \tag{6.2}$$

e' is derived from e, where $a\langle 0\rangle$ is replaced by φ_a

where P_1 is defined [1] as:

$$P_1 \equiv P^{a\langle 0\rangle,\quad a\langle 1\rangle,\quad ...,\quad a\langle d\rangle}_{\alpha(a\langle 0\rangle\,,\,i\,,\,e),\alpha(a\langle 1\rangle\,,\,i\,,\,a\langle 0\rangle\texttt{[}i\texttt{]}),...,\alpha(a\langle d\rangle\,,\,i\,,\,a\langle d-1\rangle\texttt{[}i\texttt{]})} \tag{6.3}$$

We use the same α function given in 3.8. When assigning to an element $a\texttt{[}i\texttt{]}$, where a is not currently bound to an atomic block, each index, at each depth

[1] It is not strictly necessary to define P_1 separately. We have only do so because writing P_1 in place causes the auxiliary assertion in 6.2 to extend beyond the width of the page.

in the array a is substituted by the function α. It is necessary to apply the α function at each depth because we need to modify the value at the correct index for each depth in the array. If a is bound to an atomic block then each of the ghost variables, $\varphi_a\texttt{[0]}...\varphi_a\texttt{[}n-1\texttt{]}$, are substituted by the α function. In section 6.4.1, we demonstrate the formal proof of a program that assigns to a history array which is bound to an atomic block.

As a simple example of the axiom for index-wise array assignment (6.1) we will prove the formal correctness of the following Hoare triple:

$$\{a = \langle[6\,,\,5\,,\,4]\,,\,[3\,,\,2\,,\,1]\rangle\}\texttt{a[a}\langle 1\rangle\texttt{[2]] := 7}\{a = \langle[6\,,\,7\,,\,4]\,,\,[3\,,\,5\,,\,1]\rangle\} \tag{6.4}$$

where the variable a is an index-wise history array with 3 elements and a depth of 1. The assignment causes the value 7 to be assigned to index 1 ($a\langle 1\rangle\texttt{[2]} = 1$ in the preconditions), therefore we expect that:

$$\{a\langle 0\rangle\texttt{[1]} = 7 \wedge a\langle 1\rangle\texttt{[1]} = 5\} \tag{6.5}$$

in the postconditions. All other elements in a remain unchanged.

According to our axiom for index-wise assignment (6.1) we substitute the array at each depth with an instance of the function α. Applying the index-wise assignment axiom (6.1) to the postconditions from 6.4 gives:

$$\{\alpha(a\langle 0\rangle\,,\,a\langle 1\rangle\texttt{[2]}\,,\,7) = [6\,,\,7\,,\,4] \wedge \alpha(a\langle 1\rangle\,,\,a\langle 1\rangle\texttt{[2]}\,,\,5) = [3\,,\,5\,,\,1]\} \tag{6.6}$$

The value 5 in the second instance of the function α comes from $a\langle 0\rangle\texttt{[}a\langle 1\rangle\texttt{[2]]}$ in the preconditions of 6.4, where $a\langle 1\rangle\texttt{[2]} = 1$ and $a\langle 0\rangle\texttt{[1]} = 5$. We call the preconditions from 6.4 P and the assertion in 6.6 Q. To prove the validity of the Hoare triple in 6.4 we need to show that $P \supset Q$. In P we have $a\langle 1\rangle\texttt{[2]} = 1$, therefore applying the auxiliary array assertion (3.8) to Q gives:

$$\begin{gathered}\{\alpha(a\langle 0\rangle\,,\,1\,,\,7) = [6\,,\,7\,,\,4] \wedge \alpha(a\langle 1\rangle\,,\,1\,,\,5) = [3\,,\,5\,,\,1]\} \\ \{a\langle 0\rangle\texttt{[0]} = 6 \wedge 7 = 7 \wedge a\langle 0\rangle\texttt{[2]} = 4 \wedge a\langle 1\rangle\texttt{[0]} = 3 \wedge 5 = 5 \wedge a\langle 1\rangle\texttt{[2]} = 1\}\end{gathered} \tag{6.7}$$

Clearly $P \supset Q$ and we therefore accept the Hoare triple in 6.4 as formally correct.

6.4 Array-wise arrays

We declare an array-wise array of type t with n elements and a depth of history d as:

$$\texttt{var}\ \ t\ \ \texttt{a}\langle d\rangle\texttt{[}n\texttt{]};$$

The depth specifier is placed immediately after the name to signify that the history is being stored for the entire array. The type of the above declaration is $history(\mathrm{d}) \rightarrow array(\mathrm{n}) \rightarrow t$. We can differentiate between array-wise and index-wise types since the former has the history type before the array type and the latter has the history type after the array type.

6.4.1 Formal correctness

We extend the axiom in Hoare logic for arrays (3.7) to array-wise arrays by adding substitutions for each of the historical arrays.

$$\{P_0\}\texttt{a[i] := e}\{P\} \tag{6.8}$$

We define P_0 by an auxiliary assertion which allows us to assign to array-wise arrays both inside and outside of binding atomic blocks:

$$P_0 \equiv \begin{cases} P^{\varphi_a}_{\alpha(\varphi_a\,,\,i\,,\,e')} & \text{if } a \text{ is currently bound to an atomic block} \\ P^{a\langle 0\rangle,\quad a\langle 1\rangle,\ldots,\quad a\langle d\rangle}_{\alpha(a\langle 0\rangle\,,\,i\,,\,e),a\langle 0\rangle,\ldots,a\langle d-1\rangle} & \text{otherwise} \end{cases}$$
$$e' \text{ is derived from } e, \text{ where } a\langle 0\rangle \text{ is replaced by } \varphi_a \tag{6.9}$$

The function α is defined by the auxiliary array assignment rule in 3.8. Because the history of the entire array is copied for depths 1 to d, we do not need to apply the function α to the substitution at each depth as we did for index-wise arrays (see section 6.3.2). The α function is necessary for the

assignment to the current array, since we are only assigning to a single index in the array. As we did for the index-wise axiom (6.2), the ghost variable φ_a is substituted by the α function (3.8) if a is bound to an atomic block.

We demonstrate the use of our array-wise axiom in proving the correctness of the example we used in section 6.3.2, except that here a is an array-wise array with 3 elements and a depth of 1. The Hoare triple is again given as:

$$\{a = \langle [6\,,\,5\,,\,4]\,,\,[3\,,\,2\,,\,1]\rangle\}\texttt{a[a}\langle 1\rangle\texttt{[2]] := 7}\{a = \langle [6\,,\,7\,,\,4]\,,\,[6\,,\,5\,,\,4]\rangle\} \tag{6.10}$$

The assignment statement in 6.10 causes the entire array $a\langle 0\rangle$ to be stored to history and assigns the value 7 to index 1 in the array ($a\langle 1\rangle\texttt{[2]} = 1$ in the preconditions of 6.10). Applying our axiom for array-wise history (6.8) gives:

$$\{\alpha(a\langle 0\rangle\,,\,a\langle 1\rangle\texttt{[2]}\,,\,7) = [6\,,\,7\,,\,4] \wedge a\langle 0\rangle = [6\,,\,5\,,\,4]\} \tag{6.11}$$

We call the preconditions from 6.10 P and the assertion in 6.11 Q, and attempt to prove that $P \supset Q$. In P we have $a\langle 1\rangle\texttt{[2]} = 1$. Applying the auxiliary array assertion (3.8) to Q therefore gives:

$$\begin{gathered}\{\alpha(a\langle 0\rangle\,,\,1\,,\,7) = [6\,,\,7\,,\,4] \wedge a\langle 1\rangle = [6\,,\,5\,,\,4]\} \\ \{a\langle 0\rangle\texttt{[0]} = 6 \wedge 7 = 7 \wedge a\langle 0\rangle\texttt{[2]} = 4 \wedge a\langle 1\rangle = [6\,,\,5\,,\,4]\}\end{gathered} \tag{6.12}$$

Clearly $P \supset Q$ and we therefore accept the Hoare triple in 6.10 as formally correct.

6.4.2 Implementation

In this section we deal only with the implementation of array-wise arrays in the absence of atomic blocks. We discuss how array-wise arrays are implemented in the presence of atomic blocks in section 6.4.5. Array-wise arrays can be stored as a cycle of arrays as shown in figure 6.3. The pointer ρ points to the base (first element) of the current array in the cycle. Retrieval from an array-wise array is a combination of the cyclic retrieval algorithm discussed in section 4.7.2 and standard array indexing. Retrieval from an array-wise

array can therefore be achieved in O(1) time.

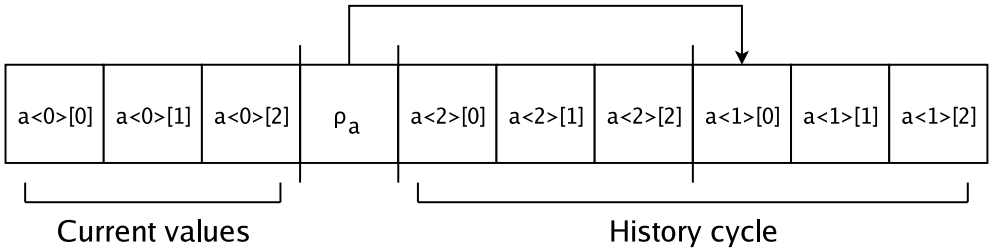


Figure 6.3: Example memory layout for an array-wise array with 3 elements and history depth of 2.

We assign values by writing to the oldest array in the cycle and then updating the cycle pointer ρ. When assigning to the oldest position in the cycle, we need to ensure that the values are correct for all elements, not just the one being assigned to. The simplest solution is to first copy the elements from the current position in the cycle to the oldest, and then assign the new value. For an array with n elements this gives us an assignment time of O(n). We require $(\text{sizeof}(t) * n * (d+1)) + \text{sizeof}(\textit{pointer})$ bytes of storage space using either the cyclic, or extended cyclic storage systems.

6.4.3 Change lists

An O(1) assignment algorithm for array-wise arrays seems unlikely since we are storing the history of the entire array, and not just a single element. We observe that, at least in the absence of atomic blocks, only a single index can differ between any two consecutive historical arrays in the cycle. Therefore we can store a fixed sized list of the changes made at each step. As discussed in chapter 2, change lists have been used by several authors to minimise the amount of memory needed to store history information. Our approach is slightly different: we store the full history information and use change list to minimise the amount of work needed to add a new item to the history. When assigning a value to an index in the array we write the changes, rather than the entire contents of the current array, over the oldest array in the cycle, before writing the newly assigned value, and finally updating the pointer.

We need to store a total of d changes, one for each depth of history. We represent the changes as an ordered list of pairs called Γ. Each change pair consists of an index and a value. Because the oldest pair in the change list is lost each time we make an assignment, it is possible to store Γ as a cycle. The change list therefore requires: $(d*(\text{sizeof}(t)+\text{sizeof}(index)))+\text{sizeof}(pointer)$ of storage space. The steps for assigning a value to an array-wise array which uses the extended cyclic storage system and a change list are outlined below:

1. Write the changes from the change list to the array at the oldest position in the cycle. The changes are written in reverse order, so the oldest change is applied first and the most recent change is applied last.

2. Write the a new change pair for the assigned value over the oldest position in the change list and update the change list cycle pointer.

3. Write the new value to the current array.

4. Update the cycle pointer ρ.

The complexity of this algorithm is $O(d)$ since there are d assignments made in step 1 above. The assignments made in steps 2 to 4 do not affect the algorithm's complexity since they require constant time, independent of both the depth of history and the number of array elements.

Clearly the performance of the $O(d)$ algorithm is only better than the performance of the $O(n)$ algorithm when $d < n$. The memory overhead for the change list makes the $O(d)$ algorithm a less viable choice for larger depths of history. A language implementation may specify that one particular algorithm is always used, or may select the algorithm for each array individually based on the values of d and n. We conjecture that the lower bound theoretical complexity for array-wise assignment is either $O(n)$ or $O(d)$.

6.4.4 Combination with atomic blocks

In the previous chapter we introduced atomic blocks as a means of temporal selection for history variables. In this section we show that we can also use

atomic blocks to assign values to multiple elements in an array-wise array before storing the history. This vastly improves the usefulness of array-wise arrays.

Consider the example function in listing 6.1 which uses an atomic block to assign all of the elements in an array-wise array's oldest history to itself. This causes the values at each depth in the array to "cycle", with the oldest history becoming the current value. For example, given an array-wise array $a = \langle[1\ ,\ 2\ ,\ 3]\ ,\ [4\ ,\ 5\ ,\ 6]\ ,\ [7\ ,\ 8\ ,\ 9]\rangle$, calling $aw\_cycle(a\ ,\ 3\ ,\ 3)$ will result in $a = \langle[7\ ,\ 8\ ,\ 9]\ ,\ [1\ ,\ 2\ ,\ 3]\ ,\ [4\ ,\ 5\ ,\ 6]\rangle$.

The function *aw_cycle* could be used with a two dimensional array-wise array to create simple animation loops. For an animation loop with k frames each of size n by m, we would use a two dimensional array-wise array of depth $k-1$ (i.e. $d = k-1$ in listing 6.1), with n rows and m columns. Each frame is stored in the array in order by assigning them inside an atomic block. Once all of the frames are in the array, the *aw_cycle* function is called to move to the next frame.

```
 1  function aw_cycle(int href a⟨⟩[], int n, int d) begin
 2    var int i;
 3    i := 0;
 4
 5    atomic(a) begin
 6      while(i < n) do
 7        a[i] := a⟨d⟩[i];
 8        i := i + 1;
 9      done
10    end
11  end
```

Listing 6.1: An example function which combines array-wise history and atomic blocks to cycle the values in an array-wise array.

We construct a proof in Hoare logic of the atomic block from listing 6.1 as follows. We call the atomic block (lines 5 – 10) T. At the beginning of T we assert that: $a = \langle A_0\ ,\ \ldots\ ,\ A_d\rangle$, where $A_0 \ldots A_d$ are the arrays at each depth of history in a. At the end of T we expect that the current value of the array

is A_d, with $A_0 ... A_{d-1}$ as the historical values. We represent this as a Hoare triple:

$$\{a = \langle A_0 \, , \, ... \, , \, A_d \rangle\} T \{a = \langle A_d \, , \, A_0 \, , \, ... \, , \, A_{d-1} \rangle\} \tag{6.13}$$

We now construct the loop invariant P for the while loop on lines 6 – 9. We call the statement on line 7 S_1 and the statement on line 8 S_2. On each iteration of the while loop the value of $a[i]$ is assigned the value of $A_d[i]$ and the value of i is incremented.

When bound to an atomic block, elements in a are saved in the corresponding element of the ghost array φ_a. At the beginning of any iteration of the while loop, where $i \geq 1$, the values of $\varphi_a[0] ... \varphi_a[i-1]$ will be equal to $A_d[0] ... A_d[i-1]$. The array a is not modified when bound to an atomic block and therefore its elements are invariant to the loop. Our loop invariant is given as:

$$P \equiv i \leq n \wedge a = \langle A_0 \, , \, ... \, , \, A_d \rangle \wedge \forall k \, | \, (0 \leq k < i) \, \varphi_a[k] = A_d[k] \tag{6.14}$$

The loop condition B is given as: $B \equiv i < n$. For the statements S_1 and S_2 we have the following Hoare triples:

$$\{P \wedge i < n\} S_1 \{P \wedge i < n \wedge \varphi_a[i] = A_d[i]\} \tag{6.15}$$

$$\{P \wedge i < n \wedge \varphi_a[i] = A_d[i]\} S_2 \{P \wedge i \leq n\} \tag{6.16}$$

We begin by showing the correctness of the statements S_1 and S_2. For clarity, we show only the relevant parts of the assertion P at each step. Applying the assignment axiom 3.5 to the postconditions in 6.16 gives:

$$\begin{gathered} \{i + 1 \leq n \wedge a = \langle A_0 \, , \, ... \, , \, A_d \rangle \wedge \forall k \, | \, (0 \leq k < i + 1) \, \varphi_a[k] = A_d[k]\} \\ \{i < n \wedge a = \langle A_0 \, , \, ... \, , \, A_d \rangle \wedge \forall k \, | \, (0 \leq k \leq i) \, \varphi_a[k] = A_d[k]\} \end{gathered} \tag{6.17}$$

By splitting the value of k in 6.17 into $k < i$ and $k = i$, we see that: $\forall k \, | \, (0 \leq k < i) \, \varphi_a[k] = A_d[k]$, which is implied by P, and $\varphi_a[i] = A_d[i]$ which is logically implied by the preconditions in 6.16. The assertion in 6.17 is therefore logically implied by the preconditions of S_2 and we therefore accept the Hoare triple for S_2 (6.16) as formally correct.

Working further backwards, we now prove the formal correctness of S_1. We replace instances of φ_a in the postconditions with the α function according to the array-wise assignment axiom (6.9). There are two instances of the α function in 6.15: the explicit instance and the instance in P. We will break the replacement of the α function into two parts. We first replace the explicit instance of φ_a in the postconditions of 6.15 and apply the auxiliary array assertion (3.8):

$$\begin{array}{c} \{\alpha(\varphi_a\,,\,i\,,\,a\langle d\rangle\texttt{[}i\texttt{]})\texttt{[}i\texttt{]} = A_d\texttt{[}i\texttt{]})\} \\ \{a\langle d\rangle\texttt{[}i\texttt{]} = A_d\texttt{[}i\texttt{]}\} \end{array} \tag{6.18}$$

which is true by definition (see 6.14). Secondly, we replace the appearance of φ_a in P (6.14):

$$\{\forall k \mid (0 \leq k < i)\; \alpha(\varphi_a\,,\,i\,,\,a\langle d\rangle\texttt{[}i\texttt{]})\texttt{[}k\texttt{]} = A_d\texttt{[}k\texttt{]}\} \tag{6.19}$$

Because k is strictly less than i we follow the second part of the auxiliary array assertion (3.8), which gives:

$$\{\forall k \mid (0 \leq k < i)\; \varphi_a\texttt{[}k\texttt{]} = A_d\texttt{[}k\texttt{]}\} \tag{6.20}$$

Therefore the fully substituted postconditions from 6.15 are:

$$\{i < n \wedge a\langle d\rangle\texttt{[}i\texttt{]} = A_d\texttt{[}i\texttt{]} \wedge \forall k \mid (0 \leq k < i)\; \varphi_a\texttt{[}k\texttt{]} = A_d\texttt{[}k\texttt{]}\} \tag{6.21}$$

which is equivalent to $\{i < n \wedge P\}$ and therefore logically implied by the preconditions of 6.15. Clearly the postconditions of S_1 (from 6.15) logically imply the preconditions of S_2 (from 6.16). Therefore according to the sequential composition axiom (3.3) and the while axiom (3.4):

$$\models \{i < n \wedge P\}\texttt{while}\ B\ \texttt{do}\ S_1; S_2\ \texttt{done}\{i = n \wedge P\} \tag{6.22}$$

The body of the atomic block only contains the while loop and is therefore already shown to be correct. We call the postconditions of 6.13 R. Applying

the auxiliary atomic block assertion (5.4) to R gives:

$$R_0 \equiv \varphi_a = A_d \wedge a\langle 0\rangle = A_0 \wedge ... \wedge a\langle d-1\rangle = A_{d-1} \tag{6.23}$$

Since $P \wedge i = n \equiv R_0 \wedge a\langle d\rangle = A_d$ we have $P \wedge i = n \supset R_0$ and therefore:

$$\frac{\{P\}\texttt{while } B \texttt{ do } S_1; S_2 \texttt{ done}\{P \wedge i = n\}\,,\, P \wedge i = n \supset R_0}{\dfrac{\{P\}T\{R_0\}}{\{P\}\texttt{atomic(a) begin } T \texttt{ end}\{R\}}} \tag{6.24}$$

Therefore we can conclude that the program in listing 6.1 is formally correct according to the pre and postconditions given in 6.13.

6.4.5 Implementation with atomic blocks

Atomic blocks do not present a problem for the O(n) algorithm since the entire current array is copied when assigning a new value. However, the change list used by the O(d) algorithm is unable to represent multiple changes between two consecutive histories.

One solution to this problem is to store a list of all of the changes made at each step rather than just a single pair. This works well in a formal sense but is difficult to implement efficiently. Because the change list is stored in memory as a cycle the number of pairs at each position must be consistent. We could allocate enough space at each position for a full list of pairs. In this case we can discard the index from each pair and simply store a list of values, requiring sizeof(t) $* dn$ bytes of storage space. This is essentially just storing a full copy of each array in the change list and requires O(dn) time to apply the changes during assignment. The O(n) algorithm is clearly superior in terms of both theoretical complexity and space requirements. Any attempt to manage a dynamic change list is likely to introduce a large runtime overhead and increase the theoretical complexity of array-wise assignment. We therefore do not discuss either the theoretical or practical implementation of a dynamic change list in this thesis.

We maintain our current change list implementation and introduce a special

change list pair called δ. In a practical implementation the pair δ can be represented by setting the pair's index to some invalid array index such as a negative number. If an array-wise array is assigned to within a binding atomic block we add a single δ pair to the change list.

If we encounter a δ pair during the application of the change list during assignment we switch to the $O(n)$ algorithm (i.e. the entire array is copied). A δ pair will remain in an array's change list for $d-1$ assignments after an atomic block. Since we can not know in advance if a change list contains a δ pair we may unnecessarily apply some of the change list before encountering the δ pair. In the best case the δ pair is the first pair in the change list. In the worst case we end up applying $d-1$ changes before overwriting them all by applying a δ pair from the last position in the change list. When a δ pair exists in the change list, we have an average assignment complexity of $O(d+n)$. In the best case, where the δ pair is the first item in the change list, we can perform the update in $O(n)$ time since the other changes do not need to be made.

We can improve on this algorithm by taking advantage of the fact the δ pairs can only remain in a change list for $d-1$ assignments after an atomic block. We no longer use the δ pairs and introduce a counter variable, called γ, for each array-wise array. The value of γ is initialised to 0 at the beginning of a program. Similarly to the cycle pointer and current value, γ may be stored independently of the array. When an array-wise array is assigned to inside an atomic block we set the value of γ to d. The change list is not modified. When assigning a value outside of an atomic block the $O(n)$ algorithm is used if $\gamma > 0$, otherwise the $O(d)$ algorithm is used. In the former case the value of γ is decremented after the assignment.

If it is possible to determine at compile time that an array-wise array is never assigned to in an atomic block then the assignment algorithm can be optimised by removing the variable γ and always using either the $O(n)$ or $O(d)$ algorithm.

6.5 Summary

In this chapter we showed that for an n dimensional array, there are 2^n possible ways to store the history. Of these methods we identified only two, which we call index-wise and array-wise, that are of practical use. Index-wise history maintains individual histories for elements, but not the relationships between histories. Array-wise history stores the history of the entire array, and therefore the relationships between histories, but does not store histories of individual elements. We outlined some examples where index-wise and array-wise history are useful.

For index-wise arrays, we require that the elements of the current array be stored contiguously in memory in order to maintain compatibility with non-history arrays. We showed that if all depths of history in an index-wise array are stored contiguously, then the optimal assignment time is $O(d)$. If only the current values are stored contiguously then assignment can be achieved in $O(1)$ time. We introduced a set of axioms for proving the formal correctness of index-wise arrays, including when they are bound to an atomic block. We demonstrated the use of these axioms with a simple example.

We also introduced a series of axioms for proving the formal correctness of array-wise history and demonstrated their use with the same example we used for index-wise arrays. We also gave a proof of an example program which used an atomic block to assign multiple values to an array-wise array before updating its history. We discussed two algorithms for assigning to index-wise arrays, both of which use a cyclic memory structure. The first, which works in $O(n)$ time, copies the entire contents of the current array to the oldest before performing the new assignment. The second stores a list of changes that have been made between the oldest and the current arrays. When assigning a new value, the oldest position is first updated with each of the changes. In the absence of atomic blocks a maximum of d changes can occur between the oldest and the current arrays. Assignment therefore takes $O(d)$ time. When atomic blocks are used it is necessary to either use the $O(n)$ algorithm or a combination of both algorithms.

It is interesting to note that while index-wise history is conceptually simpler

than array-wise history, proving the formal correctness of assignment to an index-wise array is more difficult and time consuming.

Chapter VII

User Defined Types

User defined types are one of the most useful aspects of imperative programming languages. However, in the case of structures, they are also the most difficult to efficiently implement history storage for. User defined types encompass a wide number of types in imperative languages including: structures (also known as records), unions (also known as variant records), enumerations and classes.

We ignore enumerations since they are generally implemented as a scalar type and are therefore already covered by primitive history variables (see chapter 4). We do not cover history of unions in this thesis for two reasons: It is difficult to formally reason about the correctness of unions, and unions are generally not typesafe. Modern typesafe languages, such as Java and C#, do not support unions for the latter reason.

Structures and objects are closely related, and allow several types (including other structures and objects) to be grouped together to form a new aggregate type. Classes are essentially structures that can contain functions. We do not store the history of functions and therefore focus the rest of this chapter primarily on structured types.

7.1 Representing user defined types

Structured types require both a type definition and a variable declaration. We define a structured type t using the following syntax:

```
struct t begin
  <members>
```

```
end
```

Each statement in <*members*> must be a variable declaration. A structure variable called s of type t can then be declared using the following syntax:

```
var t s;
```

We denote the elements of a structure as an n-tuple. For example a structure s with three integer members $a = 1$, $b = 2$ and $c = 3$ is denoted as: $s = (1\,,2\,,3)$. If an atomic block binds s, then it also binds all members of s.

Similar to arrays, we identify two possible methods for storing the history for structured types: We can store the history of individual members of a structure, or the history of an entire structure can be stored. We call the former approach "member-wise" history and the latter "structure-wise" history.

7.2 Member-wise history

Member-wise history occurs when we have members inside a structure that are history variables. Consider the following structure type definition:

```
struct t begin
  var int a;
  var int b⟨2⟩;
  var int c⟨3⟩;
end
```

Listing 7.1: An example structure definition with member-wise history.

The member a is a non-history variable, while the members b and c are both primitive history variables with depths of 2 and 3 respectively. Member-wise history is similar to index-wise history, except that we have a finer grain of control over the history logging. With index-wise history, every element in an array has its history stored, and the depth of history is the same for all elements. Member-wise history allows the programmer to specify individually which members are history variables.

The implementation of member-wise history is straightforward. Members which are history variables are simply stored in place in a structure. For example a structure variable of type t (from listing 7.1) could be stored in memory using the extended cyclic storage system for primitive history variables as shown in figure 7.1.

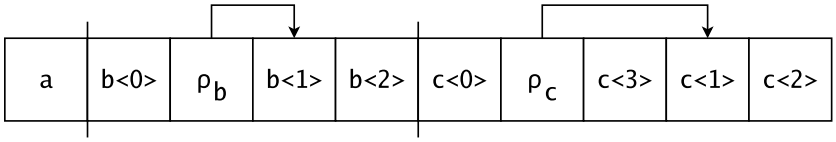


Figure 7.1: An example memory layout using the extended cyclic storage system for a structure with member-wise history.

The assignment and retrieval complexity for any history member of a structure is therefore equal to the complexity for its history type. Likewise, the formal correctness for assignment to a history member can be proven using the axioms for the type of the member.

7.3 Structure-wise history

Structure-wise history is similar to array-wise history. Whenever we assign a value to a member in a structure that has structure-wise history we store a copy of the entire structure to the history. A structure-wise history variable s of type t and depth d is declared as follows:

```
var t s⟨d⟩;
```

Note that the only difference between a structure-wise history variable declaration and a primitive history variable declaration is the type t. The members of a structure-wise history structure may themselves be history variables (i.e. member-wise history). We discuss the implications of this, and other combinations of history variables in section 7.4.

Structure-wise history can be used to implement an undo feature for aggregate types. For example the x and y coordinates of an object in a drawing program could be stored using a structure. The program in listing 7.2 shows

```
struct object_t begin
  var int x, y;
end

var object_t object⟨3⟩;

function setpos(int new_x, int new_y) begin
  atomic begin
    object.x := new_x;
    object.y := new_y;
  end
end

function undo_setpos(int level) begin
  atomic begin
    object.x := object⟨level⟩.x;
    object.y := object⟨level⟩.y;
  end
end
```

Listing 7.2: An example of using structure-wise history and atomic blocks to implement an undo feature.

example functions for setting and undoing the position of a single structure-wise variable used to represent an objects position.

The *setpos* function assigns new x and y values to the object *object*. By performing the assignments inside an atomic block, the values of both x and y are changed before the history is stored, i.e after the atomic block is exited $object\langle 1\rangle$ contains the complete previous state for the object. In the function *undo_setpos* we assign the previous value from depth *level* of each member to itself inside an atomic block (most imperative languages do not allow structures to be directly assigned to one another). For example the previous state of the variable *object* is obtained by calling $undo\_setpos(1)$.

The undo mechanism provided by structure-wise history, and by history variables in general, is a forward undo [48]. For example, if $object = \langle(1\ ,\ 2)\ ,\ (3\ ,\ 4)\ ,\ (5\ ,\ 6)\ ,\ (7\ ,\ 8)\rangle$ and $undo\_setpos(1)$ is called then the previous version of *object* is restored and its history is updated resulting in:

$object = \langle(3\ ,\ 4)\ ,\ (1\ ,\ 2)\ ,\ (3\ ,\ 4)\ ,\ (5\ ,\ 6)\rangle$. Calling $undo\_setpos(1)$ a second time acts like a redo operation, restoring the current state of *object* before the undo was performed, i.e.: $object = \langle(1\ ,\ 2)\ ,\ (3\ ,\ 4)\ ,\ (1\ ,\ 2)\ ,\ (3\ ,\ 4)\rangle$.

7.3.1 Formal correctness

Our axiom for assignment to the member m of structure-wise structure s is given as:

$$\{P_0\}s.m := e\{P\} \tag{7.1}$$

where P_0 is defined by an auxiliary assertion:

$$P_0 \equiv \begin{cases} P^{\varphi_{s.m}}_{e'} & \text{if } s \text{ is currently bound} \\ & \text{to an atomic block} \\ P^{s\langle 0\rangle.m,s\langle 1\rangle,\ldots,\ s\langle d\rangle}_{e,\ \ s\langle 0\rangle,\ldots,s\langle d-1\rangle} & \text{otherwise} \end{cases} \tag{7.2}$$

e' is derived from e, where $s\langle 0\rangle$.m is replaced by $\varphi_{s.m}$

The axioms and proofs of formal correctness for structure-wise history are similar to those for array-wise history (see section 6.4).

7.3.2 Implementation

Copying an entire structure s from one place in memory to another takes O(n) time, where n = sizeof(s), i.e the size in bytes of the structure. Therefore implementing structure-wise history using the flat storage system gives an assignment time of O(dn). The memory space requirements for the flat storage system are: sizeof(s) $*$ ($d + 1$) bytes. Using either the cyclic, or extended cyclic storage systems gives us an assignment time of O(n). As we showed for array-wise arrays (see section 6.4.2), we need to copy the entire contents of the current entry to the oldest position in the cycle before assigning a new value. Both of the cyclic storage systems require: (sizeof(s)$*$ ($d + 1$)) + sizeof(*pointer*) bytes of memory.

The change list system that we used for array-wise arrays (see section 6.4.3) is not as simple to implement for structures. With arrays we can take advantage of the fact that every element in an array is the same size. With structures

however, members can have different sizes and types. A possible solution would be to make the items in the change list large enough to hold the largest member in the structure, and store a tag value (instead of the array index) to indicate which member of the structure is being changed. The size of the change list for structure-wise history is therefore: $d * (\text{sizeof}(largest(s)) + \text{sizeof}(tag))$ bytes of storage space, where $largest(s)$ is the size in bytes of the largest member in the structure s. For dynamically typed languages, the tag value may simply be an offset, or name of the member. For statically typed languages the tag may need to store the both the offset and size of the member. Although this approach has an O(d) theoretical complexity for structure-wise assignment we expect that the practical performance will be worse than the O(d) array-wise algorithm due to the additional computations for determining the size of each change list item.

7.3.3 History arrays of structures

For either an array-wise or an index-wise array we have implicit structure-wise history if the type of the array is a structured type. For example given the following declarations:

```
struct s begin
  var int x;
  var int y;
end

var s a[10]⟨3⟩;
```

An assignment such as the following:

```
a[5].x := 10;
```

causes the history of the entire structure, not just the member x, at index 5 in the array a to be stored to history. If a is an array-wise array then the contents of all of the structures in a will be stored to history as a result of the above assignment statement.

Array-wise arrays of structures allow us to improve the functionality of the example program given in listing 7.2. If the variable *object* is instead defined as a large array-wise array, we can undo the latest change in a set of objects. Operations such as moving a group of objects can be executed as a single action by using an atomic block.

While we were able to give an O(1) assignment time algorithm for index-wise history, our current lower bound assignment time for both array-wise and structure-wise histories is either O(d) or O(n), depending on which is smaller. For applications which require very large or dynamic values of d or n, traditional approaches to history storage (see section 2.2) may be superior to a solution involving history variables.

7.4 Combining history types

With user defined types we can create aggregate types which have more than one history storage mechanism. Unfortunately this often results in redundant history storage. For example given the following declarations:

```
struct s begin
  var int a⟨1⟩;
  var int b;
end

var s x⟨1⟩;
```

The structure x has structure-wise history of depth of 1, and member-wise history for $x.a$ also of depth 1. Therefore any assignment to $x.a$ will cause the current value of $x.a$ to be stored in history twice. Consider the following Hoare triple:

$$\{x = \langle(\langle 1,2\rangle,10),(\langle 3,4\rangle,20)\rangle\}\texttt{x.a := 5}\{x = \langle(\langle 5,1\rangle,10),(\langle 1,2\rangle,10)\rangle\} \quad (7.3)$$

The assignment causes the history of both the struct x and the member $x.a$ to be updated. The value 1 ($x\langle 0\rangle.a\langle 0\rangle$ in the preconditions) has been saved as $x\langle 0\rangle.a\langle 1\rangle$ by the member-wise history update and as $x\langle 1\rangle.a\langle 0\rangle$ by the

structure-wise history update. This is similar to the problem we encountered with redundant history storage for multi-dimensional arrays (see section 6.2). In general, if an aggregate data type has more than one form of history storage then some, or all of the data that is maintained by the history logging will be redundant.

7.5 Summary

In this chapter we looked at the application of history to structured types. We identified two types of history storage called member-wise and structure-wise history, which bear close resemblance to index-wise and array-wise history (see chapter 6) respectively. Member-wise history occurs when members of a structure are history variables. The axioms for formal correctness, and the implementation details for member-wise history are those given for the type of the member.

We introduced an axiom in Hoare logic for proving the formal correctness of structure-wise history. We showed that the implementation details for structure-wise history are similar to array-wise history. The O(n) algorithm used for array-wise history can be easily modified for use with history structures. The change list based O(d) algorithm is not as simple to implement for structure-wise history, but still possible. The additional computation required for determining the size and offset of the member in a change list item will likely be detrimental to the practical performance of structure-wise history.

We showed how history arrays of structures cause implicit structure-wise history. We discussed how this could be used to improve the functionality of the undo example given earlier in this chapter. Given that our lower bound assignment complexities for array-wise and structure-wise history are linear with respect to either d or n, it is unlikely that large data structures, with large depths of history will be efficient in practice.

Finally we showed that while user defined types allowed us to construct history types which contained other history types, such as a history structure

with primitive history variables as members, it often resulted in redundant storage of historical values.

Chapter VIII

Experimental Compiler

In this chapter we discuss the design and implementation of an experimental compiler called HCC (History Capable Compiler). HCC compiles a language called HistoryC, which is based on a subset of C [1], and provides native support for history variables. A full BNF grammar for HistoryC can be found in appendix A.

8.1 Choice of compiler

We considered a number of existing compilers as a basis for our implementation work. Unfortunately the compilers we looked at were either too complex for us to gain an understanding in the time available, or too simplified for us to fully implement some features of history variables such as pointers and arrays. Because of this we decided to develop our own compiler from scratch. We were able to provide all of the necessary features in HCC at around a third the size of the source code for TinyCC [11], one of the small compilers we considered.

8.2 Design overview

HCC uses a multi-pass, frontend/backend design based on the designs described in the books: "Compilers: Principles, Techniques and Tools" (more commonly known as the "Dragon Book") by Aho, Sethi and Ullman [4] and Muchnick's "Advanced Compiler Design and Implementation" [55]. The frontend of a compiler comprises the lexer, parser, and often some intermediate representation for source programs. The backend consists of the code

generator. Thus, by changing the frontend of a compiler we can alter the language it compiles, and by changing the backend we can alter the target platform that code is generated for.

8.2.1 Frontend – HistoryC

We chose to use a subset of C as our source language for HCC since it provides us with all of the necessary features to fully test our implementation of history variables. Indirect variable access and by-reference arguments are both supported by using pointers.

Since the notation we use for history variables is not easily typeable on most keyboards we need to find an alternative. The angle bracket characters, '<' and '>', are potential candidates (Takaoka, et. al. [70] proposed their use). However, these are already used by many C based languages as relational and shift operators, and generic type delimiters. Bracha, et. al. [14] note that extra work must be taken in developing a parser for a grammar which uses the angle brackets for generic types in order to avoid parsing conflicts with other operators. To avoid potential ambiguities, and simplify the construction of a parser while maintaining a clear and readable syntax, we chose to combine two characters for each operator. We use the operators '<:' and ':>' to delimit the depth for a history variable.

In our formal notation of history variables we require that the depth be specified for the current value of a history variable. Requiring this in a programming language is somewhat cumbersome to the programmer. For example a programmer would need to edit all appearances of the current value of an existing variable if they added a depth of history to it. We therefore remove this restriction in HistoryC and allow the current value to be specified as the variable name without a depth specifier. One drawback in removing this restriction is that we lose the ability to determine if an identifier is a history variable during parsing. We solve this problem in section 8.3.2.

8.2.2 Backend – SPARC V8

We had a number of options for the target platform to generate code for: A virtual machine such as the .NET Common Language Infrastructure (CLI) or the Java Virtual Machine (JVM), a purpose built virtual machine for HCC or a native processor. Designing and implementing a virtual machine specifically for HCC would be time consuming and the resulting implementation may not translate well to existing virtual machines or processors. The JVM is an unsuitable target platform for languages other than Java [52]. This leaves us with a choice between the CLI and some native processor.

We chose to target a native processor, selecting the Sparc V8 [67] as our target platform. We chose the Sparc V8 processor because it is a cleanly implemented 32-bit RISC based machine and we had ready access to native machines running the Solaris operating system. Code compiled for the Sparc V8 is upwards compatible with the Sparc V9 processor. Generating code for a real machine rather than a virtual one allows us to look at some interesting low-level problems such as the register allocation of history variables. We will refer to the Sparc V8 architecture simply as the Sparc in the remainder of this thesis.

8.2.3 Compilation process

The frontend of HCC parses a source file and produces an n-ary abstract syntax tree (AST) representation, which is used for building the symbol table and type checking. The AST is type checked and parsed to generate a three address code representation called MIR (Medium-level Intermediate Representation). The MIR code is further transformed to a representation called LIR (Low-level Intermediate Representation) which is then passed to the backend of the compiler. The backend takes the LIR code and the nametable and generates an assembler file which can be compiled using the GNU assembler [30] to produce a binary executable. The MIR and LIR languages are based on the specifications given in [55]. In HCC, the frontend of the compiler consists of the lexer, parser and the intermediate language MIR. The backend consists of the low-level intermediate language LIR and the Sparc

code generator.

8.2.4 Register allocation

The register allocator in HCC is based on the graph-colouring approach as it is described in [55]. One notable omission is that HCC does not perform any form of register coalescing (elimination of register to register copies). This results in some redundant move instructions, especially for function arguments. We use the term "register pressure" to mean the number of registers required to allocate each of the variables in a specific piece of code. If the register pressure is too high, then some variables will be "spilled" to main memory storage. Spilling variables reduces performance.

All global variables are allocated to registers in addition to being assigned an address in main memory. Globals are loaded before each use, and stored after each definition. This ensures that the correct values are used for global variables when accessed across different function and file scopes.

8.2.5 Static pointer analysis

HCC uses a very simple form of static analysis for dealing with pointers. During the AST parse, all variables that have their address taken using the unary 'address of' operator are marked as may-aliases. May-alias variables are treated similarly to globals. They are allocated to registers as usual, but loaded from, and stored to memory when used and defined. No control flow information is taken into account and this simplified model may produce some false positives resulting in the generation of unnecessary load and store instructions.

8.2.6 Optimisations: Function inlining

HCC is, in general, not an optimising compiler. Some simple optimisations, such as constant folding, are performed since they are necessary in some cases to generate valid Sparc assembler.

HCC does provide a function inlining [15] optimisation. We use this feature to inline the history runtime library functions in our practical performance tests, to determine what effect the overhead of the runtime function calls has on performance (see section 9.5). It is not possible to use an external compiler, such as GCC, to inline the history runtime library functions. This is because HCC must be used to compile programs containing history variables and function inlining must be done during compilation. A function can be marked for inlining in HCC by specifying the inline property its prototype. HCC provides an option to disable function inlining. The original body of a function is removed upon successful inlining.

8.3 Representing history variables in a compiler

The implementation of history variables requires modifications to many parts of a compiler. This factored into our decision to not use an existing compiler as a base, as we would need to understand the entire compiler, not just some specific part of it. In this section we describe how the various subsystems of HCC were modified to accommodate history variables.

HCC implements the flat and extended cyclic storage systems for primitive history variables. Both the O(n) and O(d) assignment algorithms were implemented for array-wise arrays. Controlling the storage system used for each type of history variable is managed by a set of compiler options. We did not implement structured types and therefore HCC does not support either member-wise or structure-wise history.

8.3.1 Name table

The name table (also called the symbol table) stores information about the name and type of all of the variables and functions in a program. Four entries are made in the name table for a primitive history variable x which uses the extended cyclic storage system: The history cycle ($x\langle 1\rangle ... x\langle d\rangle$), the cycle pointer ρ_x, the current value $x\langle 0\rangle$, and the copy variable (equivalent to φ_x). Table 8.1 shows the name, type and size of each of the entries created for a primitive history variable x of type t and depth d.

Name	Meaning	Type	Size
x	The history cycle	$history(d) \rightarrow t$	sizeof$(t) * d$
.x_ptr	Cycle pointer	$pointer \rightarrow t$	sizeof$(pointer \rightarrow t)$
.x_cur	The current value	t	sizeof(t)
.x_cpy	Copy value (φ_x)	t	sizeof(t)

Table 8.1: Name table entries for a primitive history variable x of type t and depth d.

The pointer variable *.x_ptr* is marked as a may-alias when it is created since it is a reference to the current value in the cycle. The pointer, current and copy variable names are prefixed with a period to avoid namespace conflicts and direct manipulation by programmers since variable names in HistoryC (and most other C based languages) cannot start with the period character.

By creating four separate name records the compiler is able to assign different storage locations to each of the of the variables. For example x can be stored in main memory (since it will not fit in a register) while *.x_ptr*, *.x_cur* and *.x_cpy* are allocated to registers. The *.x_cpy* variable is only necessary if the variable x is modified within a binding atomic block (see section 8.4). If x is not modified by a binding atomic block in a program then the register allocator in HCC will not allocate a storage location for *.x_cpy*.

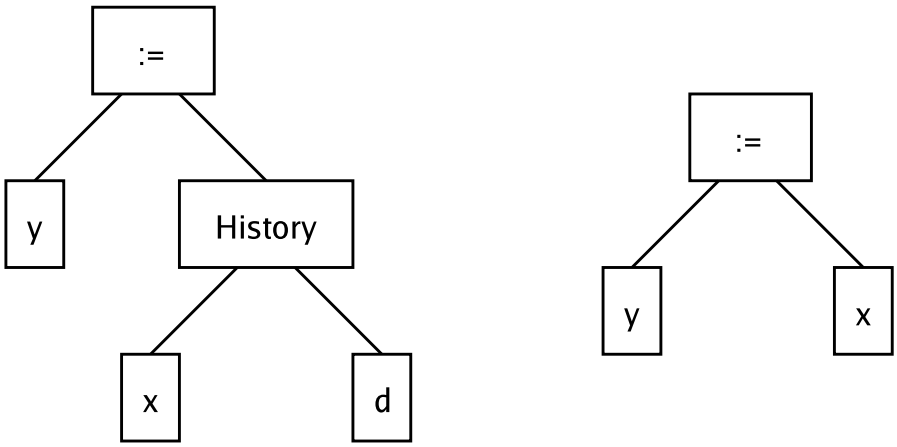
As discussed in section 4.7.5, taking the address at runtime of a primitive history variable x which is stored using the extended cyclic storage system returns the address of $x\langle 0\rangle$. When HCC encounters an expression which takes the address of a history variable x, it therefore generates code which evaluates the address of *.x_cur*. Accessing the history variable x through a pointer only modifies the variable *.x_cur* and not the history cycle.

Each of the entries shown in table 8.1 are also created for array-wise and index-wise arrays, with the types and sizes changed accordingly. For array-wise history using the O(d) algorithm, two additional variables called *.x_clist* and *.x_clist_ptr* are created for the change list and the change list pointer respectively.

8.3.2 Abstract syntax tree

We introduce a new AST node type, called "history", for specifying history depths. The history node has two children: the left child is a leaf node containing the history variable, and the right child is either a leaf or a tree representation of the history depth expression.

The parser has no knowledge of variable types (types are later assigned to nodes by the typechecker, see section 8.3.3). Given a statement such as y := x⟨d⟩, HCC can infer from context that x is a history variable and therefore generate the correct AST representation. If this assumption is incorrect (due to a programmer error for example) then an error message will be issued during the typechecking pass. For a statement such as y := x, the parser is unable to determine whether or not x is a history variable. The trees generated in each case are shown in figure 8.1. Once the nametable has been created and the types of each variable are known (based on their declarations) trees of the form shown in figure 8.1(b) are transformed to match the form shown in figure 8.1(b). The right-hand child of a history node for the current value of a history variable is the constant value zero.

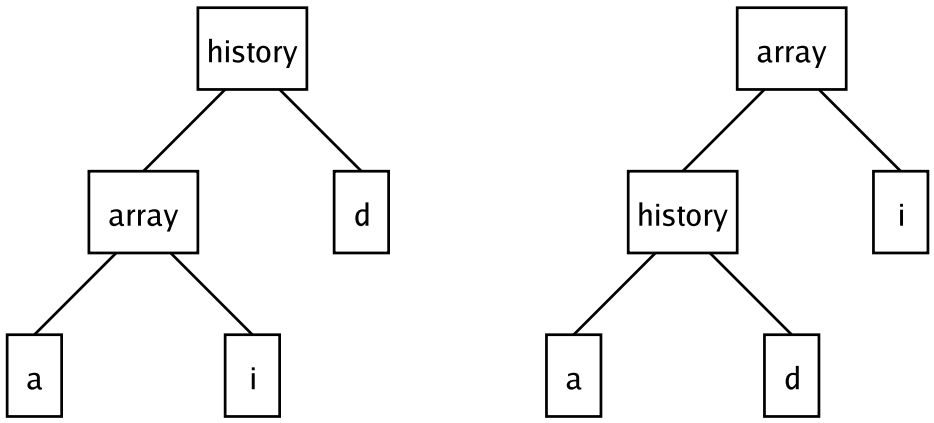


(a) y := x⟨d⟩ (b) y := x

Figure 8.1: Example AST representations for a history variable x appearing on the right-hand side of an assignment statement.

The AST representations for history array expressions are similar to those for primitive history variables. The type of history, array-wise or index-wise, for a history array can be determined by the order of the nodes in the AST as shown in figure 8.2. Array-wise history expressions are constructed with the array node as a child of the history node as shown in figure 8.2(a), while index-wise history expressions are constructed with the history node as a child of the array node as shown in figure 8.2(b). We use these AST forms because they are simple to generate when parsing a source file.

History variables appearing on the left-hand side of an assignment statement are represented in the same way as normal variables. The history nodes are not required since we cannot assign to historical values (as discussed in section 4.2).



(a) Array-wise: `a⟨d⟩[i]`

(b) Index-wise: `a[i]⟨d⟩`

Figure 8.2: Example AST representations for one-dimensional array-wise and index-wise history arrays.

8.3.3 Type information

The type system in HCC is based on the representation we introduced in section 4.1.1. The types we have given in the previous chapters for the various forms of history variables are therefore the same as those used by HCC. Types

in HCC also have an additional piece of information which indicates whether a type is an lvalue or rvalue. The typechecking pass in HCC assigns a type to each node in the AST based on the types of its children. If a type cannot be correctly assigned then an error message will be given.

The type for a history node in the AST evaluates to the same type as its left child except that the history node is an rvalue and the history type is removed (see section 4.1.1). For example in figure 8.1(a), if x of type $history(3) \rightarrow int$ and y is of type int, then the type of the node for x is $history(3) \rightarrow int$ and the history node has the type int. Therefore the assignment of $x\langle d \rangle$ to y is allowed. By using the history nodes we do not need to introduce a new set of type rules for history variables.

8.3.4 Intermediate code and runtime functions

Assignment and retrieval of history variables is managed by a set of runtime functions (see appendix B for a full list and source code). The transformation process from the AST representation to the three address code (MIR) is therefore heavily modified to accommodate history variables. The compiler must generate MIR code for calling the appropriate runtime history functions for each type of history variable. The runtime function calls can later be removed by inlining them (see section 8.2.6). We look at the practical aspects of inlining the history variable runtime functions in section 9.5.

By using runtime functions, the three address code intermediate languages do not require any specific knowledge of history variables. Therefore history variables can be implemented as an extension to the frontend of a compiler, with little or no modification required in the backend. The code generator part of the backend in HCC does have some specific code for the initialisation and debugging output of history variables, however this is not strictly necessary. HCC does not currently allow variables to have an initial defined value and therefore the initialisation of history variables is treated as a special case by the backend of HCC. Generating debugging information for history variables requires altering its type (see section 8.5). HCC currently handles this in the backend, although it should also be possible to do in the frontend.

The source code for the runtime function used to store a value to a primitive history variable using the extended cyclic storage system is shown in figure 8.1.

```
1 void __hist_p_store__(int *addr, int **ptr,
2                        int current, int depth) {
3   *ptr = *ptr - 1;
4   if(*ptr < addr) {
5     *ptr = addr + depth - 1;
6   }
7
8   **ptr = current;
9 }
```

Listing 8.1: Primitive history variable load runtime function for the extended cyclic storage system.

The cycle pointer points to the location of $x\langle 1\rangle$ in the cycle and the oldest history is alway immediately to the left (i.e. the previous word in memory). Therefore its position can be found using a simple pointer subtraction rather than the expensive computation (4.13) given in section 4.7.2. If the pointer subtraction results in an address outside of the cycle then it is set to point at the last (in terms of memory addresses) position in the cycle. Note that the new value is assigned to the variable *.x_cur* outside of the runtime function.

A noticeable limitation of the function in listing 8.1 is that it can only be used for primitive history variables of type *int*. There are at least two possible solutions to this problem. The first solution is to implement separate functions for each of the scalar types and then determine at compile time which one needs to be called for each history variable. This approach is useful for statically typed languages, but results in a much larger runtime library, especially for languages with a large number of scalar types.

The second solution is to pass information about the type of each history variable to the runtime functions. For dynamically typed languages it may be possible to pass the type itself to the function and then perform the correct operations based on the given type. For statically typed languages or languages which do not support passing types as arguments at runtime,

we could pass the size of the type (determinable at compile time) to the runtime functions and then perform block memory copies for assignment. This approach only requires a single runtime function for each operation but imposes a large amount of overhead for functions that we ideally want to be as fast as possible. We have limited HCC to deal only with integer history variables. A detailed analysis of the solution to this problem is outside the scope of this thesis.

8.4 Implementing atomic blocks

In order to simplify the implementation, atomic blocks in HistoryC are unbounded (i.e. atomic blocks bind all history variables). The implementation of atomic blocks in HistoryC is based on the approach described in section 5.4, except that a single set is maintained for the modified list μ, rather than individual sets for each atomic block, and the β sets are unnecessary since atomic blocks are not bound. If HCC encounters nested atomic blocks in a program then a warning message is issued stating that nested atomic blocks do not have any effect.

When an atomic block is entered, the compiler generates code to copy the value of *.x_cur* to *.x_cpy* for each history variable x that is bound to the atomic block. This ensures that the *.x_cpy* variable is correct if it is read from, before being assigned to within the body of the atomic block (attempting to determine this at compile time is undecidable). When an assignment to a history variable is encountered inside an atomic block, the assigned variable is added to the modified set μ. The assignment is achieved by assigning the expression on the right-hand side to the copy variable *.x_cpy*. Inside an atomic block, the current value needs to be read from *.x_cpy* rather than *.x_cur* (see section 5.2). This is handled by passing the value of *.x_cpy* as the current value argument to the runtime history retrieve function (see appendix B) when inside an atomic block. When an atomic block is exited the appropriate runtime store function is called to store the value of *.x_cpy* to *.x_cur* for each x in the modified set μ.

8.5 Debugging support

HCC provides basic support for the "stabs" debugging format [53]. This makes it possible to use a debugger, such as the GNU Debugger (GDB) [29], to examine the full history of a variable at any given point in a programs execution.

For a primitive history variable x, HCC generates three debugging symbols: x, *_x_cur* and *_x_ptr* [1]. The types for the symbols *_x_cur* and *_x_ptr* are the same as given in table 8.1. Because GDB does not understand history types, the type for x is given as: $array(d) \rightarrow t$, where d is the depth of history of x, and t is its primitive type. Printing the value of x in a debugger will display the history cycle exactly as it appears in memory. In order to make sense of the cycle, a programmer needs to first check the value of *_x_ptr* against the address of x to determine where the value of $x\langle 1 \rangle$ is located. The values of $x\langle 1 \rangle ... x\langle d \rangle$ can then be read from left to right, wrapping around the end of displayed list.

An example of an interactive debugging session using GDB is shown in figure 8.3. The lines beginning with '(gdb)' are entered by the user. Lines beginning with the '$' symbol are the responses given by GDB. The command 'print' is used to display the value of a variable. The address of a variable can be displayed by prefixing it with the '&' symbol.

```
1 (gdb) print _x_cur
2 $1 = 4
3 (gdb) p x
4 $2 = {2, 1, 3}
5 (gdb) print &x
6 $3 = (int (*)[3]) 0xffbff638
7 (gdb) print _x_ptr
8 $4 = (int *) 0xffbff640
```

Figure 8.3: An example of using GDB to print the values of a integer history variable of depth 3.

[1] The period characters in the internal names have been replaced with underscores since GDB does not recognise variable names which contain periods.

The GDB trace in figure 8.3 demonstrates the process of printing the values of an integer history variable of depth 3. The value of $x\langle 0\rangle$ is 4, as shown on line 2. The value of the pointer, shown on line 8, is 8 bytes greater than the address of the cycle shown on line 6. Because sizeof(int) = 4 on the Sparc, the value of $x\langle 1\rangle$ is the third value in the cycle shown on line 4 and therefore: $x = \langle 4\ ,\ 3\ ,\ 2\ ,\ 1\rangle$. Implementing a more user-friendly interface for examining history variables in a debugger would require modification to both the debugging format and the debugger itself. While this task is outside the scope of our research, the debugging support for history variables in HCC is a solid proof of concept.

8.6 Summary

We developed an experimental language, HistoryC, and a compiler, HCC, as a basis for investigating the practical implementation details of history variables in a compiler. HCC is a frontend/backend compiler and produces native code for the Sparc V8 architecture. HistoryC is based on the C programming language. We avoided complications in the parser and lexer by using the compound operators '<:' and ':>' as delimiters for history depth. In HistoryC we do not require that the depth be specified for the current value of a history variable as we did in our formal notation.

We discussed the implementation of history variables in HCC. Most of the required modifications are made in the frontend of the compiler. Changes were made to the name table, the abstract syntax tree and the typechecker. Minimal changes were required in the backend for initialisation of global history variables and debugging support, but we are confident that these aspects could also be handled by the frontend of a compiler.

Each history variable in HCC has four nametable entries associated with it: the history cycle, the cycle pointer, the current value and the copy value. Each of these variables can be allocated to different storage locations. The copy value is only used for assignment inside atomic blocks and is not allocated a storage location if the corresponding history variable is never bound to an atomic block. Loading and storing of history variables is managed

by a set of runtime functions. To improve the runtime performance of history variable accesses, it is possible to inline the history runtime functions. We simplified the implementation of atomic blocks by making atomic blocks unbounded. Nested atomic blocks therefore have no effect and result in a compiler warning being issued.

Finally we presented a basic approach to implementing debugging support for history variables. While our solution works, it is difficult to use. A more user friendly solution would require the debugger to be modified to support history variables.

Chapter IX

Practical Performance Analysis

In this chapter we analyse the practical performance of history variables using the experimental compiler we introduced in the previous chapter. HCC implements the flat and extended cyclic storage systems for primitive history variables. In this chapter we refer to the extended cyclic storage system as the cyclic storage system. For array-wise history we implemented both the O(n) and O(d) algorithms. HCC does not provide a full implementation of structured types. We therefore have not tested the practical performance of either member-wise or structure-wise history.

9.1 Testing platform

All of the test programs were run on a 360Mhz Sparc V9 (sun4u) machine with 1280Mb of main memory running the Solaris 9 (SunOS 5.9) operating system. Execution times were recorded using the Standard C Library clock function call, which measures processor time in microseconds. Test programs were run at least 10 times and the average time was taken to remove any effects on the performance caused by either the hardware or operating system.

In addition to HCC, we used GCC version 2.95.3 in our experiments. Although the current version of GCC at the time of writing is 4.1.1, we did not have ready access to the more recent releases on the Sparc machines available to us. For the simple programs that we are compiling we believe that GCC 2.95.3 is adequate. When running GCC in unoptimised mode we used the following compiler flags: "-ansi -pipe -mv8 -Wall". When running in optimised mode we used the same flags with the addition of the "-O2"

flag. A description of the compiler flags in GCC 2.95.3 and the optimisations it performs can be found in [28].

9.2 A brief comparison of GCC and HCC

We conducted a brief comparison of HCC and GCC in order to better understand results of experiments in which we use both compilers. Figure 9.1 shows a comparison of HCC, GCC and GCC with optimisations when used to compile the primitive history variable library (see Appendix B for the source code). The graph shows the total number of instructions, the number of memory access (load/store) instructions and the number of branch instructions generated in each case.

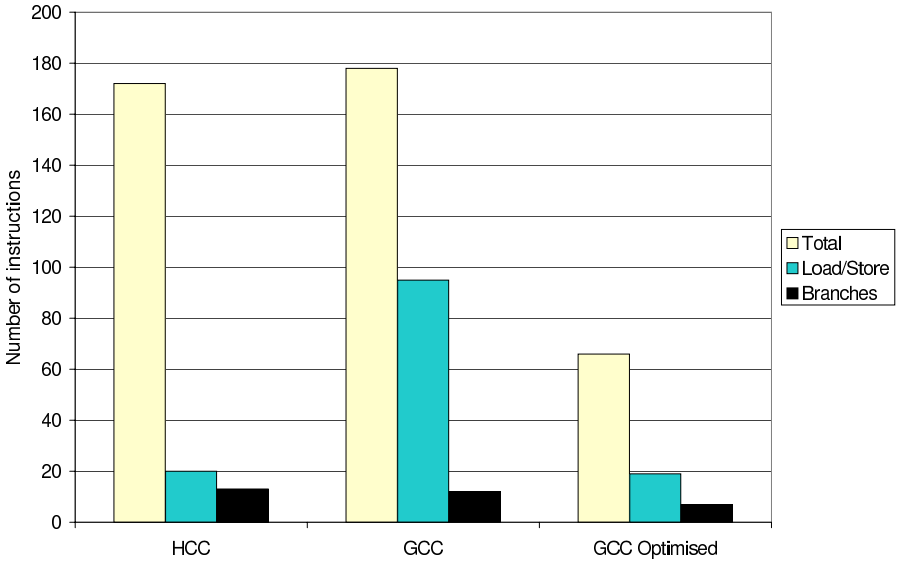


Figure 9.1: Comparison of the number of instructions generated by HCC, GCC and GCC with optimisations when compiling the primitive history variable runtime library.

HCC and GCC generate a similar number instructions in total, with approximately the same number of branch instructions. HCC generates less than a

quarter of the memory access instructions that GCC does. It appears that, at least on the Sparc, GCC does not do any real form of register allocation if optimisations are not enabled. Instead, all variables are stored in main memory, only being loaded into registers when operations are performed on them. In optimised mode, GCC generates approximately a third as many instructions as HCC, with only half as many branch instructions. HCC and optimised GCC generate similar numbers of memory access instructions.

9.3 cyclic storage system assignment performance

Our first experiment investigates the practical performance of assignment to primitive history variables using the cyclic storage system. We analyse the performance of assignment with respect to the history depth by compiling the runtime library in four different ways: Using GCC with no optimisations, using HCC with no optimisations, using GCC with optimisations and using HCC with function inlining.

Assigning to a primitive history variable using the cyclic storage system requires three assignments to be made: Assigning the current value to the oldest position in the cycle, updating the cycle pointer and assigning the new value to the history variable. We therefore expect that assignment using cyclic storage will be at least three times slower than standard assignment. Since the theoretical complexity of the cyclic storage assignment algorithm is O(1) we also expect that the performance will be consistent at each depth.

We tested the performance of assigning a constant value to integer history variables with depths ranging from 1 to 50. The assignment statements were placed in a loop which performed one million iterations. Times were recorded for the duration of the loop. As a control we timed the performance of assignment to a standard integer variable under identical test conditions. The control program was compiled with HCC. The average time for one million assignments to a non-history integer variable was 14.5ms. The results for integer history variable assignment using the cyclic storage system are shown in figure 9.2.

We expect that the performance for history variable assignment should be

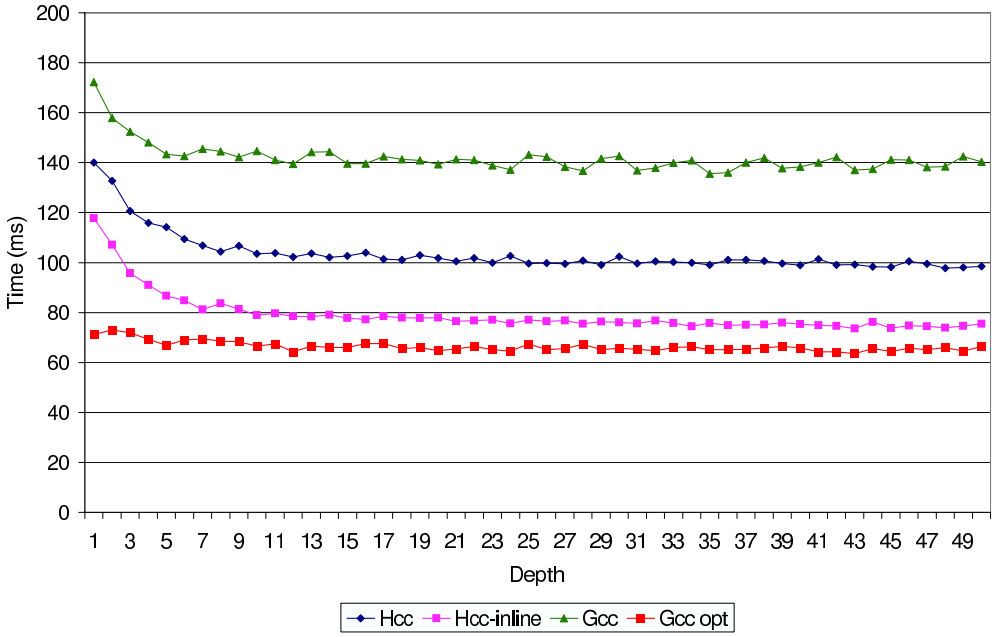


Figure 9.2: Performance comparison for primitive history variable assignment using the cyclic storage system. Times shown are for one million assignments.

consistent across all depths. However, there is a noticeable curve from depths 1 to 8 in the results for HCC, GCC and HCC with function inlining. This is caused by the additional computation (line 5 of listing 8.1) required to "wrap around" the end of the history cycle. For a history variable of depth d, the additional computation is performed once every d assignments. A history variable with a depth of 1 will therefore require the extra computation for 50% of all assignments, whereas a history variable of depth 20 only requires the extra computation for 4.76% of assignments. The optimised code produced by GCC reduces the impact of the extra computation significantly, with a difference of only 5.42ms between the time for assignment at depth 1 and the average time for depths between 9 and 50. HCC with function inlining enabled has the worst performance in this regard with a difference of 41.45ms.

The best performance for assignment using the cyclic storage system is exhibited by optimised GCC, which has an average time of 66.41ms for one million assignments. This is around 4.5 times slower than assignment to non-history variables.

Unfortunately it is not possible for us to analyse any combined benefit of function inlining and the optimisations provided by GCC. History variable accesses must be compiled by HCC, which emits assembler code containing calls to the history variable runtime library. It is not possible to subsequently inline these calls using GCC. Combining the optimisations provided by GCC with function inlining could potentially bring the performance of assignment using the cyclic storage system closer to our expected optimal result.

9.4 A comparison of flat and cyclic storage

The cyclic storage system has an assignment time of O(1). The flat storage system has an assignment time of O(d). Both storage systems have O(1) retrieval times. We anticipate that the flat storage system may be a better choice for primitive history variables which have low history depths for two reasons: The flat storage system does not require the additional internal variables *.x_ptr* and *.x_cur*, therefore lowering the register pressure, and the runtime functions for flat storage are simpler than those for the cyclic storage system. In this experiment we want to determine whether or not flat storage performs better than cyclic storage for low history depths.

We first compare the retrieval times of the two storage systems. The retrieval time was tested by assigning the values of an integer history variable x, with depths ranging from 1 to 50, to a non-history integer variable y. The values of $x\langle 0\rangle$, $x\langle 1\rangle$ and $x\langle d\rangle$ were retrieved within a loop which iterated one million times. A control program, which retrieved values from a non-history variable was also run [1]. For both the flat and cyclic storage system tests, the history variable runtime library was compiled using optimised GCC. The results of the retrieval times are summarised in table 9.1.

Retrieving a value from a non-history variables takes the same amount of time on average as assigning a value to a non-history variable since the same Sparc instruction (a single mov instruction) is used in both cases. The re-

[1] Essentially assignment and retrieval for non-history variables is the same. We use a statement such as: x := y, where x and y are integers, to test both the assignment and retrieval time for non-history variables.

	Non-history	Flat storage	Cyclic storage
Average time	14.5ms	36.5ms	58.85ms

Table 9.1: Average time taken for one million retrievals.

trieval time for a history variable using flat storage is around 2.5 times slower than retrieval from a non-history variable. Retrieval using the cyclic storage system is around 4 times slower than non-history retrieval. Therefore, if a history variable is to be primarily used for retrieval of values then the flat storage system will perform better than the cyclic storage system.

We then compared the assignment times for the two storage systems. Using the same test program from our previous experiment we compiled executables which used both the flat storage system and the cyclic storage system. The results for each are shown in figure 9.3.

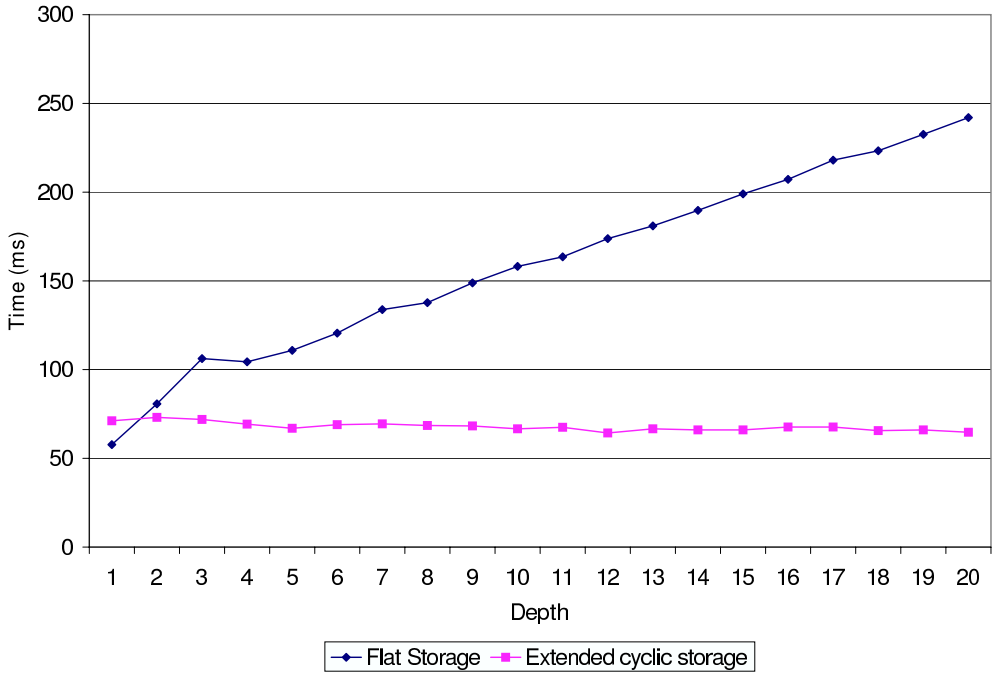


Figure 9.3: Comparison of the assignment times for the flat and cyclic storage systems. Times shown are for one million assignments.

The flat storage system only performs better than the cyclic storage system at

depth 1. At depth 2 the times are similar, with only 7.61ms difference for one million assignments. Flat storage exhibits a linear decrease in performance time with respect to the depth of history, and is almost 4 times slower than cyclic storage at depth 20.

At depth 1 the flat storage system is clearly superior: It uses less memory, has a lower register pressure, and performs better for both assignment and retrieval. For depths greater than 20, the cyclic storage system is more often a better choice due to its O(1) assignment time. The choice of storage system when the depth is between 1 and 20 is more difficult and based on outside factors. For example, if low memory consumption and register pressure are a priority then the flat storage system is a preferable choice. If the number of assignments to a history variable is significantly larger than the number of retrievals, or the assignment performance is a priority then the cyclic storage system is a better choice.

Allowing the programmer to specify which storage system should be used for each variable (i.e. using a special syntax) would offer a greater amount of control over the performance of primitive history variables. However, in the interests of simplicity (see section 1.5), we have not done this in HistoryC. Instead, HCC provides an option which allows the programmer to specify that all primitive history variables below a given depth should use the flat storage system. It may also be possible to use some form of static analysis to determine which storage system will perform better for a given primitive history variable. The implementation of this is beyond the scope of this thesis.

9.5 The effect of inlining on performance and code size

In our first experiment we showed that inlining the runtime function for storing history variables resulted in a significant performance gain. However, prior research [20, 22] indicates that function inlining is not always beneficial and can, in some cases, be detrimental to performance. Two of the adverse effects that can occur as a result of function inlining are increased code size, and reduced performance due to increased register pressure.

Our first experiment (see section 9.3) contained only a single call site (inside the loop) for the history store function. In this experiment we answer the question: How does inlining of the primitive history variable runtime functions affect the performance and code size of the resulting executable in the presence of multiple function call sites?

We examined the effect of inlining the history store functions for both the flat and cyclic storage systems. Our test programs consisted of a number of statements assigning a constant value to a primitive integer history variable of depth 1. We tested with the number of assignment statements ranging from 1 to 256, incremented in powers of 2. The assignments were placed inside a loop which iterated one million times. Times were recorded for the duration of the loop. Each program was compiled using HCC, both with and without function inlining enabled. The history variable runtime library was compiled using HCC in both cases. The performance results are summarised in table 9.2.

	Cyclic storage		Flat storage	
	No inlining	Inlining	No inlining	Inlining
Average time	128.24ms	109.16ms	150.9ms	123.65ms

Table 9.2: Average times for one million assignments to a depth 1 history variable with and without inlining the runtime history variable functions.

For the cyclic storage system, inlining the runtime functions gives a 14.9% increase in performance. Inlining the runtime functions for flat storage gives an 18% increase in performance. We measured code size as the total number of Sparc assembly instructions generated by HCC. The code size, with and without inlining, is similar for both the flat and cyclic storage systems. The results for the cyclic storage system are shown in figure 9.4.

The history runtime library is statically linked with executables. When inlining, the bodies of inlined functions are removed after function inlining is complete (see section 8.2.6). Therefore the code size of the non-inlined executables is higher when the number of assignments is low (less than 16) due to the additional function bodies present in the code. When the number of

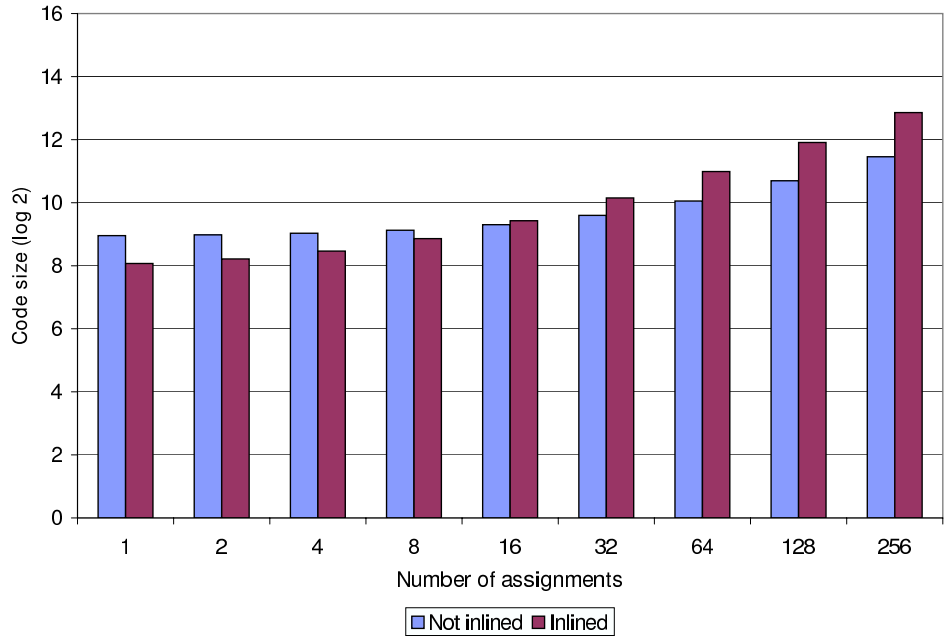


Figure 9.4: Code size, with and without function inlining, using the cyclic storage system.

assignments is 16 or greater, inlining results in larger code size. The non-inlined version has a linear increase of 9 instructions per assignment, while the inlined version increases by 28 instructions per assignment.

If an optimising compiler is able to reduce the number of instructions generated after function inlining to less than the number of instructions required to make the function call then clearly inlining the history runtime library functions is beneficial in all situations. If this not the case, then the runtime library functions should only be inlined when optimising for speed. If the compiler is optimising for memory space efficiency, then the function inlining should be disabled to reduce the number of instructions generated.

9.6 Index-wise history assignment performance

In this experiment we measure the practical performance of assignment to an index-wise array using the O(1) algorithm described in section 6.3. The test programs consist of a loop, which iterates one million times, containing

a statement which assigns a constant value to the first element of an index-wise array. Times were recorded for the duration of the loop. The test programs were compiled with the depth of the index-wise array ranging from 1 to 100, and sizes of 1, 10, 50 and 100 elements. All of the test programs were compiled with HCC. The history variable runtime library was compiled with optimised GCC. The results are summarised in figure 9.5.

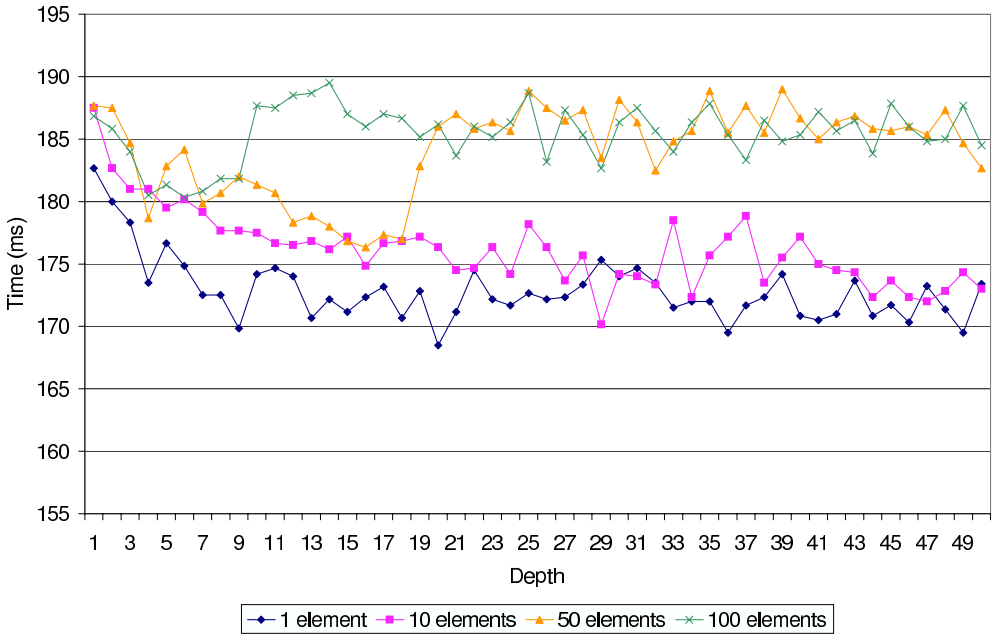


Figure 9.5: Performance of index-wise array assignment.

A control program, which assigned a constant value to the first element of a non-history array, was also created. The control program was run with array sizes ranging from 1 to 100 elements. The average time taken for one million assignments to a non-history array is 20.00ms. The average time over all of the index-wise tests we performed was 181.83ms. Therefore assignment to an index-wise array is, on average, 9 times slower than assignment to a non-history array. While index-wise arrays are essentially an array of primitive history variables the runtime performance of the index-wise store function is worse than the primitive history store function due to the additional calculations required to find the correct index in the current value and cycle pointer arrays.

As we saw in our first experiment, the assignment performance steadily increases from depth 1 and becomes roughly constant at around depth 8. Again, this is caused by the additional computation when the pointer wraps around the end of the cycle. We notice that the average times for array sizes of 50 and 100 are much higher than than the average times for the arrays of size 1 of 10. For the array with 100 elements, the time taken decreases between depths 1 and 8 and then immediately rises to an average of 188.29ms. The array of size 50 rises from an average of 180.82ms between depths 1 and 19, to an average of 188.26ms for depths 20 and above. This is caused by the requirement of additional Sparc instructions to load large constants (outside of the range -4095 – 4095, see [67]). On the Sparc the size of the history cycle variable for an integer index-wise array with 100 elements and a depth of 10 is: $100 * 10 * 4 = 4000$ bytes. Because the Sparc has a minimum stack size of 96 bytes, this results in stack offsets larger than 4095 and therefore additional instructions are required. This effect will occur at depth 1000 for an array with one element, depth 100 for a 10 element array and at depth 20 for an array with 50 elements. A conventional non-history array will require large constants for offsets if it has around 1000 elements.

9.7 O(d) vs O(n) algorithms for array-wise history

In this experiment we compare the practical performance of assignment to an array-wise array using the O(d) and O(n) algorithms we have developed (see section 6.4.2). In this experiment we answer the question: Given an array-wise array of size n and depth d which algorithm will provide the best practical performance?

The test programs contained a statement assigning a constant value to the first index of an array-wise array. The assignment statement was placed in a loop which iterated one million times. Times were recorded for the duration of the loop. Test programs for both algorithms were compiled using an array-wise array with depths ranging from 1 to 100 and a fixed size of 100 elements, and sizes ranging from 1 to 100 with a fixed depth of 100. All of the test programs were compiled with HCC. The array-wise runtime functions were

compiled using GCC with optimisations.

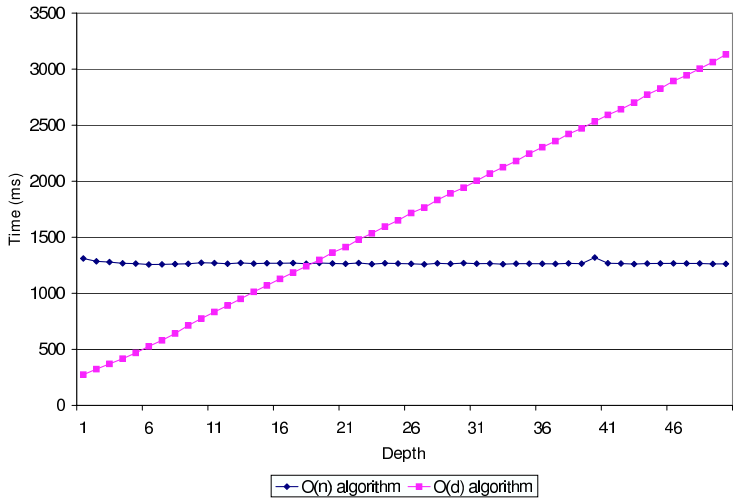


Figure 9.6: Comparison of the assignment times for the O(n) and O(d) algorithms for array-wise history at a fixed size of 100 elements.

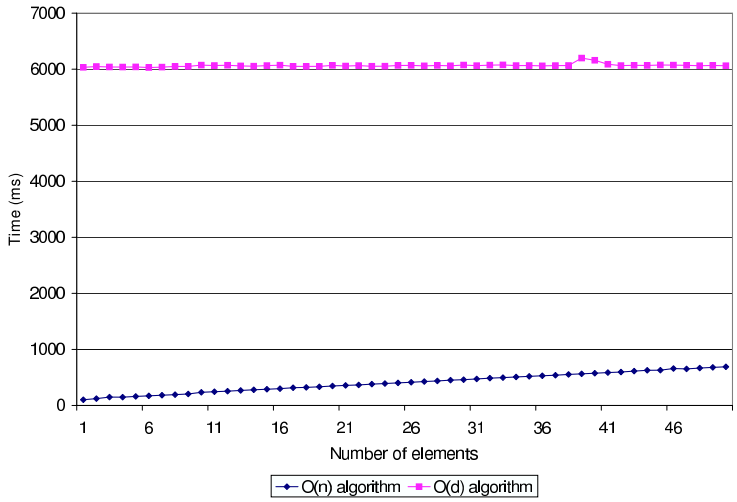
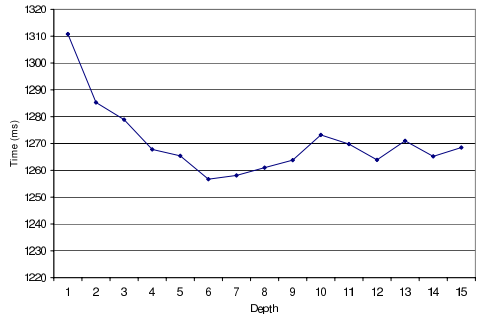
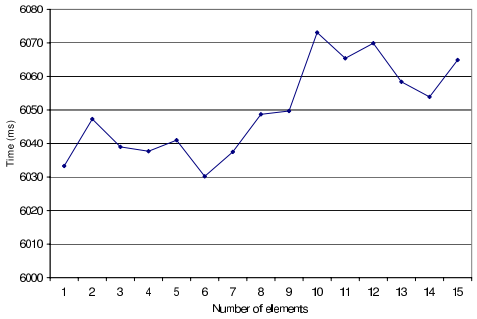


Figure 9.7: Comparison of the assignment times for the O(n) and O(d) algorithms for array-wise history at a fixed history depth of 100.

Figure 9.6 shows the performance of the algorithms with respect to history depth. The performance of the algorithms with respect to the array size is shown in figure 9.7. From the graphs we can see that the O(d) algorithm



(a) Effect of the cycle wrap computation on low depths for the O(n) algorithm.



(b) Effect of switching to large constants at d = 100, n = 10 for the O(d) algorithm.

Figure 9.8: Performance anomalies for the array-wise O(n) and O(d) algorithms.

only performs better when the depth is very low. At a fixed depth of 100, the O(n) algorithm will provide better performance even for very large array sizes. In our previous experiment we gave the average time for one million assignments to a non-history array as 20.00ms. Therefore the O(n) algorithm is at least 5 times slower standard array assignment. The O(d) algorithm is at least 13.5 times slower.

The performance anomalies discussed in our earlier experiments can also be seen for array-wise arrays. The effect of the cycle wrap computation (see section 9.3) can be seen in then performance of the O(n) algorithm, as shown in figure 9.8(a). The O(d) algorithm clearly shows the effect of switching to large constants that we encountered for index-wise arrays (see section 9.6). This effect is shown in figure 9.8(b).

By taking the average slope and approximate y-intercepts, we can use a pair of formulas to predict their runtime performance of each algorithm:

$$t_n = 90.49 * 11.71n \quad (9.1)$$

$$t_d = 216.24 * 58.56d \quad (9.2)$$

Given an array-wise array of depth d and size n, if the value of t_d is less than t_n then the O(d) algorithm will provide better runtime performance, otherwise

the O(n) algorithm should be used. The values used in these formulas need to be determined empirically for each target platform and implementation of array-wise arrays. If the expected runtime performance for a given array is similar for the O(d) and O(n) algorithms then the O(n) algorithm is a preferable option due to the lower memory requirements.

9.8 The Fibonacci Sequence

In section 4.4.1 we showed that history variables could be used to implement a simple iterative function for calculating numbers in the Fibonacci sequence. In this experiment we compare the practical performance of a Fibonacci sequence implementation which uses history variables with one which uses non-history variables. For the implementation using history variables, each Fibonacci number is calculated as shown on line 8 of listing 4.1. In our non-history implementation we replace this line with the following code:

```
t := f + f1;
f1 := f;
f := t;
```

where the variables $f, f1$ and t are non-history integer variables. The variable $f1$ stores the previous value of f, and t is a temporary variable used to store the current Fibonacci number. Using each program we calculated the first hundred Fibonacci numbers. In order to obtain useful results we placed the calculation inside a loop which iterated one million times. We tested the history variable implementation using both flat and cyclic storage. The history variable runtime library was compiled using GCC with optimisations. The results of each program are summarised in table 9.3.

	Non-history	Flat storage	Cyclic storage
Average time	1.77s	13.08s	19.64s

Table 9.3: Average time taken to calculate the first 100 Fibonacci numbers one million times.

The non-history implementation requires 3 assignments and 4 retrievals to calculate each Fibonacci number, while the history variable implementation requires 1 assignment and 2 retrievals of a depth 1 history variable. Calculation of the Fibonacci numbers using the flat storage system is 7.5 times slower than using non-history variables. Using the cyclic storage system results in the calculation being 11 times slower than using non-history variables.

From the results obtained in our previous experiments we may expect the performance of the history variable implementations to be much higher, i.e. only 4 – 5 times slower than the non-history implementation. With the non-history variable implementation, the code above for calculating a Fibonacci number can be directly represented in both our intermediate code MIR and Sparc assembly using one add, and two move instructions. This means that the add instruction performs 2 retrievals and an assignment, and the both of the move instructions perform an assignment and a retrieval.

The non-history implementations do not map directly to MIR code because of the necessity of the function calls for handling assignment and retrieval of the history variable f. The history variable Fibonacci calculation is represented in MIR as:

```
%t1 := history_load(f⟨0⟩)
%t2 := history_load(f⟨1⟩)
%t3 := %t1 + %t2
history_store(%t3, f)
```

The names $\%t1, \%t2$ and $\%t3$ are temporary storage locations which are later mapped to registers. The addition instruction appears in both the history variable and non-history variable implementations. The history variable implementation therefore has 2 history retrievals and 1 history assignment compared to 2 normal assignments in the non-history implementation. Although the history variable implementation has less assignments and retrievals overall, the presence of the function calls prevents any of these being combined in a single instruction as we saw for the non-history implementation.

Using the cyclic storage system it is possible to optimise retrievals of depths 0 and 1 by eliminating the runtime function calls. If the retrieval depth can

be determined at compile to be 0, then the history load function can be replaced with a normal load of the variable *.x_cur*. Retrieval of depth can be similarly optimised by replacing the runtime function with a dereference of the *.x_ptr* variable. Implementing these optimisations improves the time for cyclic storage to 8.55ms, only 5 times slower than the non-history variable implementation. We stated in chapter 1 that our primary goal for history variables is to simplify the task of storing the history of objects in a computer program. Clearly the history variable implementation of the Fibonacci sequence is simpler and more intuitive than the non-history version, with only a small loss of performance for our optimised version.

9.9 Summary

In this chapter we tested the practical performance of primitive history variables and arrays. We did not examine the practical performance of structure history since HCC does not have a complete implementation of structures at the time of writing. We expect that the performance of structure-wise history will be similar to array-wise history due to the similarity of the algorithms used (see section 7.3.2).

We tested the practical performance for assignment to each of the storage systems we implemented for primitive history variables and array history. A comparative summary of the performance of our history variable assignment algorithms and their non-history equivalents is given in table 9.4. We also tested the retrieval times for flat and cyclic storage and showed them to be respectively 2.5 and 4 times slower than retrieval of non-history variables.

History storage type	Comparison with non-history equivalent
Cyclic storage, O(1)	Average of 4.5 times slower
Flat storage, O(d)	At least 4 times slower
Index-wise, O(1)	Average of 9 times slower
Array-wise, O(n)	At least 5 times slower
Array-wise, O(d)	At least 13.5 times slower

Table 9.4: Summary of history variable assignment performance.

We discovered that the performance of assignment to history variables which are stored in cyclic buffers steadily increases over the first few depths. We showed that this is caused by the additional computations which need to be performed if the cycle pointer wraps around the end of the cycle during assignment. For history variables with low depths, this wrap around happens frequently and causes a decrease in the assignment performance. A second performance anomaly we discovered was, due to the large size of the cycle variables for history arrays, large constants were required for indexing history arrays with only moderately large sizes or history depths. On the Sparc large constants require additional instructions therefore causing an decrease in performance. We showed that this effect occurred on the Sparc when $4dn \geq 4000$. Large constants are only required for conventional arrays with more than 1000 elements.

For array-wise history we showed that unless the value of d is very large, the $O(n)$ algorithm provides better performance. We empirically determined a pair of formulas which can be used to predict the runtime performance of each of the algorithms.

Finally we compared the practical performance of an implementation of the Fibonacci sequence using history variables (from listing 4.1) with an iterative implementation which uses non-history variables. Our results were surprising, showing the history variable implementation to be up to 11.5 times slower than the non-history version. We described, and implemented two optimisations for history variables which improved the performance to only 5 times slower.

Chapter X

Conclusions and Future Work

In this thesis we have presented the semantics, formal correctness and implementation details of history variables in an imperative programming language. We developed an experimental language called HistoryC, which implements most of the aspects of history variables that we introduced. We also developed an experimental compiler for HistoryC called HCC which allowed us to test our implementation strategies and analyse the practical performance of history variables.

10.1 Fulfilling our implementation goals

In section 1.5 we outlined a set of goals for not only implementing history variables in an imperative programming language, but also ensuring that they were efficient and practical to use. In this section, we discuss our success in fulfilling each of these goals.

10.1.1 Complete implementation

Our goal for completeness states that history variables should be ubiquitous in a programming language, i.e. it should be possible for any variable type in a language to be declared as a history variable. None of the existing languages we discussed which support some form of history mechanism have a complete implementation. History storage was limited to either specific variable types or specific situations in a program.

In this thesis we have presented the semantics, formal correctness and implementation of history for scalar types, pointers, strings, arrays and structured

types. The application of history to these types should prove sufficient for a complete implementation in most procedural imperative programming languages. We also introduced a new programming construct, called an atomic block, which can be used to temporarily suspend the history logging of a variable. Our experimental language, HistoryC, supports atomic blocks and history storage for most of the types we discussed in this thesis.

Our research focused on the implementation of history variables in a procedural imperative language rather than an object orientated one, and as such we have not investigated the application of history to classes. We note that classes are very similar to structures, and that much of our work on history structures could also be applied to classes. We therefore consider this goal to have been achieved.

10.1.2 Compatibility with non-history variables

In order to provide a seamless integration of history variables in a programming language we stated that history variables should be compatible with non-history variables. It should be possible to use the current or historical values of a history variable anywhere that the value of a non-history variable of the same type could be used.

Our approach to solving this problem was to start by making history variables a different type to non-history variables. We say that a history variable x of depth d and type t has the type: $history(d) \rightarrow t$. Each particular value of the history variable, $x\langle 0\rangle ... x\langle d\rangle$, is of type t and therefore immediately compatible with other non-history variables of type t. We prevent assignments to the historical values by introducing a semantic rule which states that the current value of a history variable is an lvalue, while its historical values are rvalues. This initial approach allows us to freely assign between history and non-history variables and pass either the current or historical values of a history variable to functions expecting by value arguments of type t.

Compatibility between history variables in the presence of pointers, reference types and by-reference function arguments proved more difficult and

we outlined two possible semantics: Indirect assignment to a history variable updates its history, and indirect access to a history variable modifies only the current value. We showed that the first semantic is more ideal, but limited by incompatibilities with existing pointer types. For the first semantic we introduced a new pointer type, called a history pointer, which can be used to reference an entire history variable of any depth. We showed that the implementation of the first semantic can be accomplished using existing pointer types. However, the cyclic memory buffers which we used to store history variables could not be used in the presence of pointers, since the memory addresses of a history variable were not fixed. We solved this problem by introducing the extended cyclic storage system, which has a fixed memory address for the current value of a history variable. We showed that the two semantics are not mutually exclusive, and can be implemented simultaneously in a language.

Our implementation of history variables gives a high level of compatibility between history and non-history variables, even in the presence of pointers. We therefore consider this goal to have been achieved.

10.1.3 Correctness

We stated that our proposed semantics for history variables should be formalised and verifiably correct. We achieved this goal by presenting axioms in Hoare logic (see chapter 3), which were used as a formal specification of the semantics of each of the history types we introduced in this thesis. Using these axioms, we provided example proofs of some interesting example programs.

Our axiom for assignment to a primitive history variable is a simple extension of Hoare's original assignment axiom. We illustrated the use of this axiom by giving a proof of a program which calculates the first n Fibonacci numbers using a history variable (see listing 4.1). In chapter 5 we introduced an axiom to formalise the semantics of singly bound atomic blocks. Our atomic block axiom uses a ghost variable for maintaining the current value of the bound history variable inside an atomic block. We demonstrated our atomic block

axiom with a number of example programs, including one with nested atomic blocks.

In chapter 6 we introduced axioms for assignment to both index-wise and array-wise history arrays. As discussed in chapter 3, assignments to array indicies are mapped by the array assignment axiom to a function called α. For index-wise arrays we showed that it was necessary to map the substitutions at each depth of the array to α. Array-wise history stores entire copies of the array to history and we therefore showed that it was only necessary to map the assigned index to the α function for the array at depth 0. In both cases we showed that it is necessary to map occurrences of elements in the ghost variable φ_a to the function α for any array a that is bound to an atomic block. We demonstrated the use of our axioms, for both index-wise and array-wise history, in the proof of a complex array assignment. We also gave a formal proof of a program which combined array-wise history with an atomic block to build a function which can cycle the history of an array-wise array (see listing 6.1).

In chapter 7 we introduced two forms of history storage for structured types called member-wise and structure-wise history. We showed that member-wise bears a close resemblance to index-wise history and structure-wise is similar to array-wise history. It was not necessary to introduce formal axioms for member-wise history since the appropriate axiom for the type of the member can be used. The axiom we gave for structure-wise assignment is a simple modification of the standard axiom for structure assignment given in chapter 3. Because the axioms for history structures are similar to those for history arrays we did not provide any example proofs in chapter 7.

We have provided axioms in Hoare logic for atomic blocks, and each of the history types we have introduced in this thesis. These axioms can be used to verify the formal correctness of programs which use history variables and, where appropriate, we have given example proofs. We therefore conclude that our goal for correctness has been achieved.

10.1.4 Simplicity

We stated that history variables should have simple syntax and semantics and should be easy for an experienced programmer to gain an understanding of in a reasonably short period of time. Our goal for compatibility between history and non-history variables greatly simplifies the use of history variables since, in many cases, they are treated the same as their non-history counterparts. For example assignment, mathematical operations and the passing of variables as arguments to a function are syntactically identical for both history and non-history variables.

A formal study to quantify the simplicity of history variables is outside the scope of this thesis. We are confident, however, that our proposed syntax and semantics for history variables would fare well in such a study, and therefore consider this goal to have been achieved.

10.1.5 Performance

We stated that history variables should be efficient in terms of memory usage and that accessing a history variable should not be significantly slower than accessing a non-history variable. In this thesis we analysed the performance of history variables in both theoretical and practical terms.

For primitive history variables we have succeeded in this goal. Both the cyclic and extended cyclic storage systems we introduced in 4 have theoretical assignment and retrieval times of O(1), with a minimal overhead in memory storage. We implemented the latter storage system in our experimental compiler. Empirical tests showed that assignment to history variables is 4.5 times slower than assignment to non-history variables. Retrieval was determined to be 4 times slower than retrieval from a non-history variable.

We introduced an algorithm for index-wise arrays, which had a minimal memory storage overhead, and theoretical assignment and retrieval times of O(1). Empirical tests showed assignment to an index-wise array to be, on average, 9 times slower than assignment to non-history arrays. The practical results for primitive history variables and index-wise arrays are acceptably

fast, and we conjecture that a combination of inlining the history variable runtime functions and performing compiler optimisations could further improve their practical performance. We consider our goal for performance to have been achieved for primitive history variables and index-wise arrays.

We showed that the implementation details for array-wise history and structure-wise history are similar, and developed two algorithms for assignment. We tested the practical performance of array-wise arrays using both of these algorithms. The first algorithm takes $O(n)$ time, where n is the number of elements in the array, or the size in bytes of the structure. The second algorithm takes $O(d)$ time, where d is the depth of history. Our empirical tests on assignment time showed these algorithms to respectively be at least 5 and 13.5 times slower than their non-history counterparts. We conjectured that the lower bound for array-wise and structure-wise history assignment is either $O(n)$ or $O(d)$, whichever is smaller. For history arrays and structures of larger sizes and history depths, the practical performance is poor, particularly for the $O(d)$ algorithm. We therefore do not consider our performance goal to have been fully achieved for array-wise and structure-wise history.

10.2 Future work

We consider each of our five goals to have been at least partially accomplished. However, there is still much room for improvement in our research. Two large areas for continuing research are the need for a formal study to be undertaken to assess the practicality and simplicity of history variables from a users point of view, and the improvement of the practical performance of history variables, particularly for array-wise and structure-wise history.

Our practical performance for primitive history variables and index-wise arrays is acceptable, but not optimal. Investigation into optimisations, both human and compiler based, for the history variable runtime functions may help improve their practical performance. If our conjecture for the optimal assignment time of array-wise and structure-wise history is correct then continuing research should focus on improving the implementation of our algorithms. Our $O(d)$ algorithm, in particular, performs poorly in practice. If

we are incorrect, then further research may uncover algorithms with better theoretical complexities.

History variables as a debugging feature appears interesting. In this thesis we described the addition of basic debugging support for history variables to our experimental compiler. Our approach is neither complete, nor user friendly. We showed that improving our approach would require modification of the target debugger. A possible extension to history variables as a debugging feature would be to store not only the past value of a variable, but also the time, or line of code where the previous value was assigned. History variables could also be used as the basis for the implementation of a reversible debugger as discussed in section 2.1.6.

Finally, it remains to be seen how effective history variables are in solving real world programming problems. Continuing research could investigate the use of history variables as a basis for the history storage in the various situations we described in chapter 2.

Appendix A

HistoryC Grammar

HistoryC is based on a subset of C. The grammar for HCC is based on the Degener's Yacc grammar [24] for ANSI C. The major differences in HistoryC from C are:

- HistoryC uses the "var" keyword before variable declarations and the "func" keyword before function declarations.
- The assignment operator in HistoryC is handled separately from the other operators. Assignments in HistoryC must appear as a standalone statement, i.e. it is not possible to have an assignment inside an expression.
- HistoryC supports a native string type. Strings are immutable and have a string descriptor which stores the length and a pointer to the string data. Most operations on strings, such as concatenation, result in a new string being created and the string descriptor being updated.
- There are no side effect operators, such as C's post increment, in HistoryC. We therefore have not investigated the effects of side effect operators on history variables and atomic blocks.

Below is the full grammar for HistoryC in standard BNF notation:

```
<program_chunk>
  ::= ε
    | <program_chunk> <global_declaration>
    | <program_chunk> <func_declaration>
    | <program_chunk> <function>
```

```
<declarations>
  ::= ε
    | <declarations> <declaration>

declaration
  ::= var <type> <identifier_list> ;
    | var struct <struct_member_list> <declarator> ;

<global_declaration>
  ::= var <type> <identifier_list> ;
    | var struct <struct_member_list> <declarator> ;

<struct_member_list>
  ::= { <struct_members> }

<struct_members>
  ::= <type> <declarator> ;
    | <struct_members> <type> <declarator> ;

<identifier_list>
  ::= <declarator>
    | <identifier_list> , <declarator>

<type>
  ::= void
    | char
    | string
    | int
    | <identifier>

<declarator>
  ::= <declarator2>
    | <pointer> <declarator2>

<declarator2>
  ::= <identifier> <history_declaration>
    | <identifier> <history_declaration>
      <array_declaration> <history_declaration>

<history_declaration>
  ::= ε
    | <: <number> :>

<pointer>
  ::= *
    | <pointer> *

<array_declaration>
  ::= <array_declaration> [ <number> ]
    | [ <number> ]
```

```
<identifier_arguments>
  ::= <expression>
    | <identifier_arguments> , <expression>

<func_inline>
  ::= ε
    | inline

<func_declaration>
  ::= func <func_inline> <type> <pointer>
      <identifier> <func_params_decl> ;
    | func <func_inline> <type> <identifier>
      <func_params_decl> ;

<func_params_decl>
  ::= ( )
    | ( ... )
    | ( <param_decl_list> )

<param_decl_list>
  ::= <type> <declarator>
    | <param_decl_list> , <type> <declarator>

<function>
  ::= <type> <pointer> <identifier>
      <func_params_decl> <block>
    | <type> <identifier> <func_params_decl> <block>

<block>
  ::= { <declarations> }
    | { <declarations> <statement_list> }

<statement_list>
  ::= <statement>
    | <statement_list> <statement>

<statement>
  ::= <assignment_statement> ;
    | for ( <assignment_statement> ; <expression> ;
      <assignment_statement> ) <block>
    | while ( <expression> ) <block>
    | do <block> while ( <expression> ) ;
    | if ( <expression> ) <block>
    | return <expression> ;
    | <atomic_block> <block>

<atomic_block>
  ::= atomic
    | atomic ( <atomic_list> )
```

```
<atomic_list>
  ::= <identifier>
    | <atomic_list> , <identifier>

<assignment_statement>
  ::= <expression> = <expression>
    | <expression>

<expression>
  ::= <or_expression>

<base_expression>
  ::= <identifier>
    | <number>
    | <char_literal>
    | <string_literal>
    | ( <expression> )

<postfix_expression>
  ::= <base_expression>
    | <postfix_expression> [ <expression> ]
    | <postfix_expression> <: <expression> :>
    | <postfix_expression> ( )
    | <postfix_expression> ( <identifier_arguments> )
    | <postfix_expression> . <identifier>
    | <postfix_expression> -> <identifier>

<unary_expression>
  ::= <postfix_expression>
    | <unary_op> <unary_expression>

<cast_expression>
  ::= <unary_expression>
    | [ <type> ] <unary_expression>
    | [ <type> <pointer> ] <unary_expression>

<mult_expression>
  ::= <cast_expression>
    | <mult_expression> <mulop> <cast_expression>

<add_expression>
  ::= <mult_expression>
    | <addop mult_expression>
    | <add_expression> <addop> <mult_expression>

<relational_expression>
  ::= <add_expression>
    | <relational_expression> <relop> <add_expression>

<and_expression>
  ::= <relational_expression>
```

```
    | <and_expression> && <relational_expression>

<or_expression>
  ::= <and_expression>
    | <or_expression> || <and_expression>

<unary_op>
  ::= *
    | &

<addop>
  ::= +
    | -

<mulop>
  ::= *
    | /

<relop>
  ::= <
    | <=
    | >
    | >=
    | ==
    | !=
```

Appendix B

History Variable Runtime Library

Below is the source code for the history variable runtime library used by HCC. All of the functions can be compiled using either HCC or any ANSI C compiler.

```
#ifdef __HCC__
# define __VAR var
#else
# define __VAR
#endif

/*
 * Primitive history variable store. Extended cyclic storage
 */
void __hist_p_store__(int *addr, int **ptr, int current, int depth) {

  /* Overwrite oldest history and update pointer */
  *ptr = *ptr - 1;
  if(*ptr < addr) {
    *ptr = addr + depth - 1;
  }

  **ptr = current;
}

/*
 * Primitive history variable load. Extended cyclic storage.
 */
int __hist_p_load__(int *addr, int current, int *ptr, int depth, int history) {
  if(history == 0) {
    return current;
  }

  /* Find the position in the cycle */
  ptr = ptr + (history - 1);

  if(ptr >= addr + depth) {
    ptr = ptr - depth;
```

```
  }

  return *ptr;
}

/*
 * Primitive history variable store. Flat storage.
 */
void __hist_fp_store__(int *addr, int depth, int value) {
  __VAR int *base;

  base = addr;
  addr = addr + depth;

  while(addr > base) {
    *addr = *(addr - 1);
    addr = addr - 1;
  }

  *base = value;
}

/*
 * Primitive history variable load. Flat storage.
 */
int __hist_fp_load__(int *addr, int depth) {
  return *(addr + depth);
}

/*
 * Initialise an index-wise array. Takes O(n) time.
 */
void __hist_iw_init__(int *addr, int **ptr, int size) {
  __VAR int i;

  for(i = 0; i < size; i = i + 1) {
    *(ptr + i) = addr + i;
  }
}

/*
 * Index-wise store.
 */
void __hist_iw_store__(int *addr, int **ptr, int *current, int index,
                       int size, int depth, int value) {
  __VAR int *base, **idx, *cidx;

  /* Base of the current index */
  base = addr + (size * index);

  /* Common subexpression elimination. HCC doesnt do this */
```

```
  idx = ptr + index;
  cidx = current + index;

  /* Assign the new value and update the cycle */
  *idx = *idx - 1;
  if(*idx < base) {
    *idx = base + (depth - 1);
  }

  /* Assign the value */
  **idx = *cidx;
  *cidx = value;
}

/*
 * Index-wise load.
 */
int __hist_iw_load__(int *addr, int *ptr, int *current, int index,
                     int depth, int history, int size) {
  __VAR int *pos, *base;

  /* Get the copy value if the history is 0 */
  if(history == 0) {
    return index;
  }

  base = addr + (size * index);
  pos = ptr + index;

  pos = pos + (history - 1);
  if(pos >= base + depth) {
    pos = pos - depth;
  }

  return pos;
}

/*
 * Initialise an array-wise array for use with the O(d) algorithm.
 */
void __hist_daw_init__(int *addr, int **ptr, int *clist, int **clist_ptr,
                       int depth) {
  __VAR int i;

  *ptr = addr;
  *clist_ptr = clist;

  /* Initialise the change-list */
  for(i = 0; i < (depth * 2); i = i + 2) {
    *(clist + i) = 999;
  }
```

```
}

/*
 * Store a value to an array-wise array using the O(n) algorithm
 */
void __hist_naw_store__(int *addr, int **ptr, int *current,
                        int depth, int size, int index, int value) {
  __VAR int i, *oldest;

  /* Find the oldest history */
  *ptr = *ptr - size;
  if(*ptr < addr) {
    *ptr = addr + ((depth - 1) * size);
  }

  /* Copy the current array to the oldest */
  for(i = 0; i < size; i = i + 1) {
    *(*ptr + i) = *(current + i);
  }

  /* Assign the new value */
  current[index] = value;
}

/*
 * Store a value to an array-wise array using the O(d) algorithm
 */
void __hist_daw_store__(int *addr, int **ptr, int *current, int depth,
                        int size, int *clist, int **clist_ptr, int index,
                        int value) {
  __VAR int i;

  /* Find the oldest history */
  *ptr = *ptr - size;
  if(*ptr < addr) {
    *ptr = addr + ((depth - 1) * size);
  }

  /* Apply the change list */
  for(i = 0; i < depth; i = i + 1) {
    /* Apply change item. An index of 999 represents a delta pair */
    if(**clist_ptr != 999) {
      *(*ptr + **clist_ptr) = *(*clist_ptr + 1);
    }

    /* Move change list pointer */
    *clist_ptr = *clist_ptr + 2;
    if(*clist_ptr >= clist + (depth * 2)) {
      *clist_ptr = clist;
    }
  }
```

```
  /* Write the assignment to the change list */
  *clist_ptr = *clist_ptr - 2;
  if(*clist_ptr < clist) {
    *clist_ptr = clist + ((depth * 2) - 2);
  }

  **clist_ptr = index;
  *(*clist_ptr + 1) = value;

  /* Assign the new value */
  *(current + index) = value;
}

/*
 * Load a value from an array-wise array. Used for both the O(d) and
 * the O(n) array-wise algorithms.
 */
int *__hist_aw_load__(int *addr, int *current, int *ptr, int depth,
                      int size, int index, int history) {
   if(history == 0) {
     return current;
   }

   ptr = ptr + ((history - 1) * size);

   if(ptr >= addr + (size * depth)) {
      ptr = ptr - (size * depth);
   }

   return ptr;
}
```

References

[1] *American National Standard Programming Language C, ANSI X3.159-1989*, December 14 1989.

[2] G. Abowd and A. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317 – 342.

[3] Hiralal Agrawal, Richard A. Demillo, and Eugene Spafford. An execution backtracking approach to program debugging. Technical Report SERC-TR-22-P, 1990.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[5] Suad Alagic and Michael A. Arbib. *Design of Well-Structured and Correct Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1991.

[6] Apple Computer Corporation, Cupertino, CA. Programmer's guide to MacApp. Available from: `http://developer.apple.com/documentation/mac/MacAppProgGuide/MacAppProgGuide-2.html`, 1986.

[7] Krzysztof R. Apt. Ten years of Hoare's logic: A survey - part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431 – 483, 1981.

[8] James E. Archer Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1 – 19, January 1984.

[9] Phil Bagwell. Fast functional lists, hash-lists, deques and variable length arrays. In *Implementation of Functional Languages, 14th International Workshop, Madrid, Spain*, pages 34 – 50, 2002.

[10] R. M. Balzer. Exdams: Extendable debugging and monitoring system. In *AFIPS Spring Joint Computer Conference*, pages 567 – 580, 1969.

[11] Fabrice Bellard. TCC: Tiny C compiler. Available from: `http://fabrice.bellard.free.fr/tcc/`, 2001.

[12] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102 – 126, 2000.

[13] Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. Submitted to the SPACE '04 Workshop, available from: `http://www.cs.mdx.ac.uk/staffpages/r_bornat/papers/separation_and_aliasing.pdf`.

[14] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java programming language: Participant draft specification. 2001. `http://java.sun.com/developer/earlyAccess/adding_generics/`.

[15] P. P. Chang and W. W. Hwu. Inline function expansion for compiling C programs. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 246 – 257, New York, NY, USA, 1989. ACM Press.

[16] Maurice Clint. Program proving: Coroutines. *Acta Informatica*, 2(1):50 – 63, March 1973.

[17] Maurice Clint. On the use of history variables. *Acta Informatica*, 16(1):15 – 30, August 1981.

[18] Jonathan J. Cook. Reverse execution of Java bytecode. *The Computer Journal*, 45(6):608 – 619, 2002.

[19] James W. Cooper. *Java Design Patterns: A tutorial.* Addison Wesley Longman, Inc., 2000.

[20] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software - Practice and Experience*, 21(6):581 – 601, 1991.

[21] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualisation of programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125 – 150, April 2000.

[22] Jack W. Davidson and Anne M. Holler. Subprogram inlining: a study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89 – 102, 1992.

[23] Jaco de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall International Inc., London, 1980.

[24] Jutta Degener. ANSI C Yacc grammar. Available from: `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`, 1995.

[25] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109 – 121, New York, NY, USA, 1986. ACM Press.

[26] Eclipse Foundation. Eclipse - an open development platform. Available from: `http://www.eclipse.org/`.

[27] W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. A temporal model for multi-level undo and redo. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 31 – 40, New York, NY, USA, 2000. ACM Press.

[28] Free Software Foundation. GCC documentation. Available from: `http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.ps.gz`.

[29] Free Software Foundation. GDB: The GNU project debugger. Available from: `http://www.gnu.org/software/gdb/documentation/`.

[30] Free Software Foundation. GNU binutils. Available from: `http://www.gnu.org/software/binutils/`.

[31] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, October 2004.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Resusable Object-Orientated Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[33] Fabio Grandi and Maria Rita Scalas. HoTQuel: A history-oriented temporal query language. In *IEEE European Computer Conference, Bologna (Italy)*, 1991.

[34] Fabio Grandi, Maria Rita Scalas, and Paolo Tiberio. A history-oriented temporal SQL extension. In *Next Generation Information Technologies and Systems*, 1995.

[35] S. Greenberg and I. H. Witten. How users repeat their actions on computers: principles for design of history mechanisms. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 171 – 178, New York, NY, USA, 1988. ACM Press.

[36] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Peer-to-Peer Communications, 1996.

[37] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[38] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848 – 894, 1999.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 580, 1969.

[40] C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335 – 355.

[41] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1):1 – 6, 1997.

[42] John H. Howard. Proving monitors. *Communications of the ACM*, 19(5):273 – 279, 1976.

[43] Michael JasonSmith and Andy Cockburn. Get a way back: evaluating retrieval from history lists. In *CRPITS '03: Proceedings of the Fourth Australian user interface conference on User interfaces 2003*, pages 33 – 38, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[44] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13 – 26, 1997.

[45] Stephen H. Kaisler. *Interlisp: The Language and its Usage*. John Wiley and Sons, Inc., 1986.

[46] Tomasz Kowaltowski. Data structures and correctness of programs. *Journal of the ACM*, 26(2):283 – 301, April 1979.

[47] Francis C. M. Lau and Atul Asthana. Yet another history mechanism for command interpretors. *SIGPLAN Not.*, 19(3):51 – 56, 1984.

[48] George B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50 – 87, January 1986.

[49] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules of the programming language Euclid. *Acta Informatica*, 10(1):1 – 26, March 1978.

[50] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226 – 244, 1979.

[51] John W. Macvey. *Time Travel: A Guide to Journeys in the Fourth Dimension*. Scarborough House/Publishers, 1990.

[52] E. Meijer and J. Gough. Technical overview of the common language runtime. Technical report, Microsoft, 2000. Available from: `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.

[53] Julia Menapace, Jim Kingdon, and David MacKenzie. The "stabs" debug format. Available from: `http://developer.apple.com/documentation/DeveloperTools/gdb/stabs/stabs_toc.html`.

[54] Microsoft Corporation. Undoengine class. Available from: `http://msdn2.microsoft.com/en-us/library/system.componentmodel.design.undoengine(VS.80).aspx`, 2006.

[55] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[56] MySQL AB. MySQL AB: The world's most popular open source database. Available from: `http://www.mysql.com/`, 1995.

[57] Paul J. Nahin. *Time Travel: A Writer's Guide to the Real Science of Plausible Time Travel*. Writer's Digest Books, 1997.

[58] David V. Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173 – 1214, 2001.

[59] Robert Osterlund. The Pikt script language. Available from: `http://pikt.org/pikt/ref/ref.html`, 1998.

[60] Mark H. Overmars. Searching in the past II - general transforms. Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands, 1981.

[61] Susan Owicki. A consistent and complete deductive system for the verification of parallel programs. In *STOC '76: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 73 – 86. ACM Press, 1976.

[62] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279 – 285, 1976.

[63] Python Software Foundation. The Python programming language. Available from: `http://www.python.org/`, 1990.

[64] Chet Ramey. The GNU history library. Available from: `http://tiswww.tis.case.edu/~chet/readline/history.html`, 2002.

[65] Douglas J. Santry, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Veitch. Elephant: The file system that never forgets. In *Workshop on Hot Topics in Operating Systems*, pages 2 – 7, 1999.

[66] Rok Sosic. History cache: hardware support for reverse execution. *Computer Architecture News*, 22(5):11 – 18, December 1994.

[67] Sparc International, Inc. *The Sparc Architechture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992. Available online from: `http://www.sparc.org/standards/V8.pdf/`.

[68] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32 – 41, 1996.

[69] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 132 – 143, New York, NY, USA, 1977. ACM Press.

[70] Tadao Takaoka, Michael Maclean, and Bruce McKenzie. Introduction of history to variables (short communication). *Software: Practise and Experience*, 12(6):595 – 597, 1982.

[71] Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Arie Segev, and Richard T. Snodgrass (Eds). *Temporal Databases: Theory, Design and Implementation.* Benjamin-Cummings, 1993.

[72] Warren Teitelman. Interlisp reference manual. Technical report, Xerox Co., Palo Alto Research Center, Palo Alto, California, 1978.

[73] John Tucker and Jeffery Zucker. *Program correctness over abstract data types, with error-state semantics.* Elsevier Science Publishers B.V., Amsterdam, 1988.

[74] Jennifer Vesperman. *Essential CVS.* O'Reilly and Associates, Inc., June 2003.

[75] David Wall. Register windows vs register allocation. In *PLDI '88: Proceedings of the SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 67 – 78, New York, NY, USA, 1988. ACM Press.

[76] John Zukowski. *John Zukowski's Definitive Guide to Swing for Java 2.* Apress, 6400 Hollis St, Suite 9, Emeryville, CA 94608, 1999.