Incremental Garbage Collection for Haskell

An MEng Computing Final Year Project Report

by Andrew Cheadle

Department of Computing,
Imperial College,
180 Queen’s Gate,
London,
SW7 2BZ

Version 1.00, June 18, 1999
Copyright ©1999 Andrew Cheadle and Imperial College. All rights reserved.

This document is free and is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Abstract

This report presents an incremental garbage collection scheme for the lazy functional language, Haskell, and documents its design, implementation and benchmark results against other well known algorithms. The collector is based on the ‘While and Field’ non-stop garbage collector [While and Field, 1992] for the ‘Spineless Tagless G-machine’ [Peyton Jones, 1992], an abstract machine for lazy functional languages. Whilst GHC, the ‘Glasgow Haskell Compiler’ [Peyton Jones et al., 1992], the runtime system of which is the platform for the implementation of these schemes, is a stock hardware implementation of the Spineless Tagless G-machine, it deviates a little from the abstract architectural concepts, in that the objects are not so tagless. The net result of efficiency and optimisation decisions made in the design of the new runtime system for GHC [Marlow and Peyton Jones, 1998] has meant that it is no longer possible to implement the While and Field collector without modification. It is the new scheme, with the required modifications, that is presented here.
Acknowledgements

I would like to thank my supervisor Tony Field for his excellent help and guidance with this project, Simon Peyton Jones, Simon Marlow and Sighbjorn Finne (all at Microsoft Research, Cambridge) for their help in enlightening me as to the weird and wonderful workings of the GHC compiler and its installation; and Pasquale Malacaria for a very informative lecture course on the semantics of functional programming languages, without which some of the concepts in this project would have been even harder to grasp!

A special thanks goes to Simon Marlow for the patience and help he extended to me during the implementation phase of the project; for, as chief ‘GHC runtime system guru’, he was my main source of reference for the intricacies of both the runtime system and garbage collection for GHC.

Finally, thanks must also go to the MEng ”posse” for encouragement and sanity preservation, namely Sanjay Buhdia, Øyvind Eikeland, Dave Gardner, Tom Hansen, Chris Harris, Tony Hickson, Tony Pareira, Andy Sadler and Rajesh Sinha.
Preface

Any suggestions for the addition, removal or alteration in any of the content of this document should be forwarded to the author or project supervisor. Feedback is encouraged from other researchers who have evidence to support or refute the results presented here.

Andrew Cheadle <a.cheadle@doc.ic.ac.uk>
Dr. Tony Field <ajf@doc.ic.ac.uk>

Department of Computing,
Imperial College of Science, Technology and Medicine

The current version of this document is available on:
www.doc.ic.ac.uk/~amc4/GHCSelfScavGC.html
Contents

Abstract iii
Acknowledgements v
Preface vii

1 Introduction 1
1.0.1 Project Goals ................................................. 4
1.0.2 Report Structure ........................................... 5

2 Functional Language Implementation 7
2.1 The Lambda Calculus ............................................ 8
2.1.1 λ-notation ................................................... 8
2.1.2 The Fixed Point Theorem .................................. 9
2.1.3 Reduction .................................................. 10
2.2 Representing and Evaluating a Program ....................... 11
2.2.1 The SECD machine ......................................... 14
2.2.2 Graph Reduction .......................................... 16
2.2.3 The G-machine ............................................. 19
2.3 Introducing GHC,
   The Glasgow Haskell Compiler .................................. 25
2.3.1 The Goals of GHC ........................................... 25
2.3.2 The Structure of the Compiler ............................ 25
2.3.3 The Spineless Tagless G-machine ......................... 27

3 Garbage Collection and Basic Collectors 31
3.1 The Reference Counting Algorithm ............................ 31
3.1.1 Lazy Garbage Collection ........................................ 33
3.1.2 Single-bit Reference Counting .......................... 34
3.1.3 Weighted Reference Counting ............................ 35
3.2 Mark - Sweep Garbage Collection ......................... 36
3.3 Mark - Compact Garbage Collection ......................... 38
  3.3.1 Pointer Threading ........................................ 39
  3.3.2 Jonkers' Compaction Algorithm ....................... 40
3.4 Copying Garbage Collection .................................. 41
3.5 Generational Garbage Collection ............................ 42
3.6 Concurrent and Incremental Garbage Collection ........... 44

4 Baker's Algorithm .................................................. 47
  4.1 Algorithm Description ...................................... 47
  4.2 Pseudo-code Algorithm ..................................... 51
  4.3 Performance Implications ................................... 53

5 GHC Runtime System and Heap Objects ....................... 55
  5.1 An Overview of the New Runtime System ................. 55
    5.1.1 The Storage Manager .................................. 55
    5.1.2 The Scheduler ......................................... 59
    5.1.3 The Interpreter ....................................... 59
  5.2 The GHC Heap ................................................ 60
    5.2.1 Universal Representation of Closures ............... 61
    5.2.2 GHC Heap Objects ..................................... 62
  5.3 The GHC Stack ............................................... 65

6 The While and Field Garbage Collector ....................... 67
  6.1 Algorithm Description ...................................... 67
  6.2 Pseudo-code Algorithm ..................................... 69
  6.3 Performance Implications ................................... 75

7 An Incremental Self-Scavenging Garbage Collector for GHC 77
  7.1 The Spineless Not-So-Tagless G-Machine ................. 77
  7.2 Algorithm Description ..................................... 78
CONTENTS

7.2.1 Eliminating the Read-Barrier — The Self-Scavenging Object .............................................. 78
7.3 Pseudo-code Algorithm ..................................................... 80
7.4 GHC Specific Considerations ........................................... 83
   7.4.1 Pointed Objects — Not the Only Type of GHC Heap Object ........................................... 84
7.5 Performance Implications ............................................... 87

8 Implementation and Testing ........................................... 91
   8.1 The Self-Scavenging Collector ........................................ 91
      8.1.1 Occurrence of a Heap Overflow ............................. 92
      8.1.2 Initiation of the Garbage Collector ..................... 94
      8.1.3 Evacuating an Object ....................................... 95
      8.1.4 Collector Scavenging ....................................... 100
      8.1.5 Mutator Scavenging ........................................ 101
      8.1.6 Termination of Garbage Collection ...................... 102
      8.1.7 The Simulated Scavenge Queue ............................ 103
      8.1.8 Scavenging Tospace ....................................... 104
      8.1.9 Handling Large Objects .................................... 106
      8.1.10 Handling Weak Pointers ................................. 106
      8.1.11 Scavenging Static Objects ............................... 107
      8.1.12 Closure Entry ........................................... 109
   8.2 The Baker Collector .................................................. 110
      8.2.1 Evacuating an Object ....................................... 110
      8.2.2 The Read Barrier .......................................... 111
      8.2.3 Scavenging Tospace ....................................... 113
   8.3 Testing ............................................................. 113

9 Benchmarking .............................................................. 115
   9.0.1 The Benchmark Programs ....................................... 115
   9.0.2 ‘Raw’ Execution Times ....................................... 116
   9.0.3 Benchmark Results ............................................ 117
   9.0.4 Analysis of Results .......................................... 118
10 Conclusion and Further Work 123
  10.1 Conclusion .................................................. 123
  10.2 Further Work ............................................... 125
      10.2.1 Improving Performance ............................ 125
      10.2.2 Incremental Stack Scavenging .................... 126
      10.2.3 Incremental Scavenging of Unpointed Objects .... 127
      10.2.4 Implementation of an Incremental Collector into GHC-
      4.02 ............................................................ 128
List of Figures

2.1 The compilation route — High-level language to target language. .......................... 20

3.1 Pre-threaded structure. ............................... 39
3.2 Post-threaded structure. ............................... 40

4.1 Heap at initiation of garbage collection: root set = \{O_1, O_2\}. ............................... 49
4.2 Semi-spaces after root evacuation. ............................... 49
4.3 Semi-spaces after one call to \texttt{mutator\_create()} and \(k = 4\) scavenges. ............................... 50
4.4 Semi-spaces after two calls to \texttt{mutator\_create()}. Garbage collection completes before \(2k = 8\) scavenges. The memory allocated to the Fromspace objects may now be freed. ............................... 50

5.1 The components of the GHC runtime system. ............................... 56
5.2 GHC closure layout. ............................... 61

6.1 Closure layout required by the While and Field collector ............................... 68

7.1 Closure layout required by the self-scavenging collector ............................... 79

8.1 Basic flow of execution for the self-scavenging collector ............................... 92
8.2 Control flow of \texttt{GC\_incremental\_Scavenge} ............................... 100
8.3 Closure layout used by the Baker collector ............................... 110
# List of Tables

9.1 User-times for the (non-profiled) stop and copy collectors operating over the selected test programs. .......... 118
9.2 User-times for the (non-profiled) incremental collectors operating over the selected test programs. .......... 118
9.3 Number of garbage collections that occur during execution of the test programs. ................................. 119
9.4 Profiled user times for the Cirsim test program. ......... 119
9.5 Profiled user times for the Compress test program. ....... 119
9.6 Profiled user times for the Wave4main test program. .... 119
9.7 Profiling overheads for the test programs on varying $k$. .. 120
9.8 Average pause time between mutator executions for varying values of $k$. ........................................... 120
Chapter 1

Introduction

Haskell is a purely functional, non-strict (lazy) language that has now become a de facto standard for the non-strict functional programming community [Peyton Jones et al., 1992], and there are now many compilers available. Since its birth in 1987 it has evolved significantly [Hudak, Peterson and Fasel, 1997], and is currently documented by the Haskell 98 report.

Two of the more advanced Haskell systems are the Glasgow Haskell Compiler (GHC) and the Hugs interpreter. GHC, originally developed at the Department of Computing Science, University of Glasgow, is under constant development and the current version release is 4.02. The Hugs interpreter also undergoing constant development at the Department of Computer Science, Yale University, provides an environment in which Haskell programs can be developed and quickly tested before compilation. Such a compilation is usually performed by GHC, producing a native machine executable that incorporates a runtime system allowing the program to run, as a stand-alone application without depending on an aiding environment. It is for this reason, that the authors of the two systems now work much more closely together with the goal of providing support for mixed interpreted/compiled execution of Haskell programs. Both GHC-4.02 and Hugs98 (the current release version of Hugs) implement the standards defined in the Haskell 98 report.

Over the past few years many investigations have been carried out into the comparison of different garbage collection algorithms. It is now widely
accepted that there is no single scheme that outperforms any other but that
the algorithm employed is not only application dependent but also the one
that gives the most acceptable performance as a result of compromise of
the costs involved. Examples of the costs that need to be considered are
the total execution time of the program, the delay period in the resump-
tion of the user program (the mutator process, as opposed to the garbage
collector process), the proportion of the time that is spent garbage collect-
ing, the memory overheads which result from the invocation of the collector
and of course the amount of memory that may be utilised as heap space at
any one time. It is not just the need to make the above compromises that
makes quantifiable comparisons between the collectors so difficult, but also
architectural considerations, such as the speed of the processor, the pres-
ence of virtual memory, the size and number of caches and the efficiency
with which the developer implemented the collection algorithm. In short,
true comparisons are only really going to be achieved by implementing the
algorithms for comparison within the same environments and applied to the
same applications.

At least three different garbage collection schemes have been employed
during GHC’s lifetime, a one-space in-place compacting collector, a two-
space stop and copy collector and a generational garbage collector, for which
benchmark figures do exist ([Sansom, 1991], [Sansom and Peyton Jones,
1993]). While and Field proposed a variant on Baker’s incremental garbage
collection algorithm [While and Field, 1992] for implementation into the
Spineless Tagless G-machine running on stock hardware. The algorithm
was intended to eliminate the stop/go execution associated with bulk copy-
ing collection algorithms, allowing the system to place an upper bound on
the time taken to perform a store operation’. It was designed such that in
its implementation, Baker’s read barrier was eliminated, through the ma-
nipulation of the code pointers associated with each object closure, and as
such, it was expected to be ‘considerably more efficient than previous imple-
mentations of Baker’s algorithm’.

The move from version 3.0x to 4.0x of GHC has facilitated the changes
required for the mixed interpreted/compiled execution of Haskell programs
mentioned above and thus compatibility with Hugs. It has also allowed for
deficiencies in the old system to be rectified and new features to be added [Marlow and Peyton Jones, 1998]. Unfortunately, however, the changes made in the runtime system are such that it is no longer possible to implement a While and Field garbage collector. The pointers to code used by the garbage collector have now been eliminated from structures associated with each object closure, the existence of which are totally fundamental to the implementation of the While and Field collector — it was found that they led to garbage collection code which was both scattered and hard to optimise.

The initial project brief proposed a full implementation of the While and Field collector into GHC and its benchmarking in order to produce results that would support the preliminary investigations of While and Field. These investigations suggested that 'the collector carries a typical CPU overhead of less than 10% of a program's execution time, comparable with or better than previous figures published for hardware implementations' [While and Field, 1996]. As already mentioned, supporting this for version 4.0x, requires significant re-engineering of the compiler, which could not be achieved in the time available. Whilst the required code pointers were present in version 3.0x of GHC, its stack layout prevents implementation of such a collector.

It is worth noting, that the major redesign and partial rewrite that the compiler and runtime system had just undergone resulting in the transition from version 3.0x to 4.0x means that the re-engineering of any version of the compiler other than the latest is highly undesirable. Although a While and field collector can be expected to be significantly more efficient than comparable incremental algorithms, it is unlikely to be able to recover the reduction in garbage collection overhead would not be able to recover the reduction in efficiency of the compiler as a whole that prompted the redesign in the first place. If this is indeed the case, and it is hard to determine, then the implementation that resulted would have been of no practical use other than to substantiate While and Field's preliminary investigation.

As the language features of Haskell expand and the development of GHC advances in support, so the user base of the language has rapidly grown. Not only has it gained the crown previously held by Miranda as the leading functional programming language, but it is increasingly being employed
for software development in areas other than academia, where it has been used mainly for teaching and research purposes. The introduction of (foreign language) interfaces to both CORBA and imperative and object orientated languages as well as impressive input/output functionality provide the framework for hybrid interactive applications, which no longer just number crunch/pattern match their way to solutions of mathematical problems. As the applications become increasingly interactive, response times of the program become critical, and the period of time that elapses as a complete garbage collection occurs, pausing useful computation and user interaction, often become unacceptably long. Such applications therefore initiate the requirements for an incremental garbage collector that places an upper bound on these pause times.

1.0.1 Project Goals

The aims of the project are as follows:

- To gain an understanding of the issues involved in implementing compilers for functional languages and more specifically the runtime system that supports the compiled program.

- To evaluate the performance of the While and Field collector through the implementation and benchmarking of a variant of their collector against other algorithms, thus providing further insight into applicable garbage collection techniques for non-strict functional languages.

- To deliver an incremental garbage collector for GHC 4.01 that supports interactive applications and enforces an upper bound on pause times.

Subdivision of these aims results in the following goals:

- Presentation of introductory material on the foundations of functional programming languages and their implementation.

- Presentation of introductory material on dynamic memory management and garbage collection schemes.
• Design and implementation into GHC of a variant of the While and Field collector.

• Implementation of an incremental Baker collector.

• Comparative benchmarking of the two incremental collectors and the existing stop and copy collector.

• Suggestions for further investigation and optimisation of the work achieved including some detailed design suggestions.

1.0.2 Report Structure

This report is divided into 10 chapters:

Chapter 1 Provides an introduction to the project as a whole, to its goals, to GHC and to the While and Field collector.

Chapter 2 is a background section providing a very brief overview of the concepts and techniques involved in the implementation of functional languages. The material is not central to the project and may be safely skipped on first reading.

Chapter 3 introduces the fundamentals of dynamic memory management and the classical garbage collection algorithms.

Chapter 4 describes in detail the classical ‘Baker’ algorithm for incremental garbage collection.

Chapter 5 provides a technical overview of the components of GHC’s runtime system and defines the types of object that can exist within the GHC heap.

Chapter 6 describes the collector upon which the variant that is presented in this report is designed and why it is no longer practically implementable.

Chapter 7 details the design of an incremental self-scavenging garbage collector that is suitable for implementation into GHC.
Chapter 8 details the implementation of two incremental garbage collectors, a self-scavenging collector, and a Baker collector.

Chapter 9 presents the results of comparative benchmarking of the two collectors along with the stop and copy collector currently implemented into GHC-4.01.

Chapter 10 summarises the work achieved, whether the initial project goals have been met and provides suggestions of where the work can be extended and additional investigations performed.
Chapter 2

Functional Language Implementation

This chapter attempts to:

- Familiarise the reader with the basic theory that is the foundation to any programming language, but more specifically to functional languages.

- Briefly explain the different types of expression evaluation models and two abstract machine implementations for functional languages that have resulted, namely the SECD machine and the G-machine.

- Provide an overview of the compilation phases that a high-level language must go through, including intermediate representations and abstract machine representation, before reaching low-level machine code.

- Introduce GHC and the abstract architecture upon which it is based, namely the spineless and tagless variant of the G-machine.

It is difficult to achieve the above without explaining the concepts that follow in great detail. Certainly, there is no intention to educate the reader in the many subtleties that exist and so only an overview is presented. The interested reader is referred to [Field and Harrison, 1988] and [Peyton Jones and Lester, 1992] for further details.
2.1 The Lambda Calculus

The λ-calculus [Church, 1941], devised by Alonzo Church, provides a ‘universal’ model of computation, which means that any computation that can be expressed as a Turing machine can also be expressed using λ-calculus. It is, at some level of abstraction, at the heart of every programming language, and allows the expression of (anonymous) functions (or function abstractions), function applications and recursion. Essentially, it ‘provides a formalism for expressing functions as rules of correspondence between arguments and results’ [Hankin, 1994].

2.1.1 λ-notation

The λ-calculus uses λ-notation for the expression of rules and contains a set of axioms and rules that define how computation of terms expressed in the notation may be performed. The Backus-Naur Form (BNF) of the λ-notation is as follows:

\[
\begin{align*}
< \lambda - \text{term} > & ::= < \text{variable} > \mid \\
& \quad < \text{abstraction} > \mid \\
& \quad < \text{application} > \\
< \text{variable} > & ::= x \mid y \mid z \ldots \\
< \text{abstraction} > & ::= (\lambda < \text{variable} > < \lambda - \text{term} >) \\
< \text{application} > & ::= (< \lambda - \text{term} > < \lambda - \text{term} >)
\end{align*}
\]

A function (abstraction) is therefore defined to perform operations on the inputs to which it is applied (as an application) and yield the results of these operations in the form of outputs; the arguments may themselves be any λ-term, most usefully other abstractions or applications.

It is now convenient to introduce several terms and concepts that will be referred to throughout the remainder of this report.

- **Closed expression** — an expression in which no free variables exist.
- **Closure** — a composite structure consisting of an expression and its bindings given in the environment extant from the point of evaluation.
2.1 The Lambda Calculus

More simply, a closure represents a closed expression, which is a tuple consisting of the bound variable identifier of the \( \lambda \)-expression, the body of the \( \lambda \)-expression, and an associated environment [Field and Harrison, 1988].

- \textit{Environment} — maps values to the free variables belonging to the body of a \( \lambda \)-expression.

- \textit{Combinator} — a closed function.

- \textit{Constructor} — a function that builds data structures by binding the data elements together.

- \textit{Currying} — Because of the way in which substitution and reduction are performed, a multi-argument function of \( n \) arguments is treated one argument at a time, i.e. as a concatenation of \( n \) single-argument functions.

- \textit{Free variable} — a variable with which no value is associated.

- \textit{Partial Application} — of a function arises when an application only partially completes the arguments of that function. In both Miranda and Haskell, a partial application is created when a defined function is applied to fewer arguments than appear on the left-hand side of its definition.

- \textit{Supercombinator} — a simplified function definition with all its arguments as simple variables.

- \textit{Suspension} — an unevaluated (sub-)expression, where the argument expression is ‘frozen’ until it is needed in the evaluation of an instantiated function body. It is represented as a tuple of the expression and an environment.

2.1.2 The Fixed Point Theorem

The fixed point theorem makes the definition of recursive functions possible. For \( \lambda \)-terms \( f \) and \( x \), \( x \) is said to be the fixed point of \( f \) if \( fx = x \), that is, ‘a fixed point of a function \( f \) is simply an expression which remains unchanged
as a result of applying \( f \) to it’ [Field and Harrison, 1988]. Stated formally, the Fixed Point Theorem is:

\[
\forall F. \exists X. X = FX
\]

To represent a recursive function, which can be thought of as a function taking itself as an argument, the function is defined as a lambda abstraction in which an extra parameter is added to enable the function to be passed to itself. The fixed point combinator or \( \text{Y-combinator} \) is used to bind the function to itself, with this combinator satisfying:

\[
Yf = f(Yf)
\]

2.1.3 Reduction

The evaluation of \( \lambda \text{-terms} \) is performed by the application of two reduction rules, the alpha (\( \alpha \)-) reduction rule and the beta (\( \beta \)-) reduction rule. The alpha reduction rule allows the consistent renaming of bindings of variables:

\[
\lambda x. T \rightarrow \lambda z. \{z/x\}T
\]

The rule holds for any \( z \) which is neither free nor bound in \( T \) and \( \{z/x\}T \) is defined as ‘the substitution of \( z \) for \( x \) for any free occurrence of \( x \) in \( T \)’. The beta reduction rule allows the consistent replacement of an argument for the \( \lambda \text{-term} \)'s bound variable in its body when the application of a \( \lambda \text{-term} \) to the argument is performed:

\[
(\lambda M) N \rightarrow [M/x]N
\]

\([M/x]N\) is defined as ‘the substitution of \( M \) for \( x \) for any free occurrence of \( x \) in \( N \)’. It is worth mentioning, that the Church-Rosser theorem states that the result of chained substitutions is independent of the order in which they are performed.

The current \( \lambda \)-expression being reduced is called the \textit{redex} (short for reducible expression) and the evaluation of expressions is performed by the repeated reduction of its redexes until no more redexes exist. When a \( \lambda \)-expression cannot undergo further reduction, it is said to be in \textit{normal form}. 
2.2 Representing and Evaluating a Program

\( \beta \)-reduction is used to replace the occurrences of bound variables by argument (expression)s and \( \alpha \)-reduction is used to rename variables that would undergo a ‘name’ clash were the variable being substituted into the abstraction to have the same name as the bound variable of the abstraction.

A third type of reduction rule exists \( \delta \)-reduction rules, that are built in rules for the application of constant functions. For example, the following expression is the application of the ‘built-in’ arithmetic function + to two integers 2 and 4:

\[
+ 2 4
\]

The above expression can be reduced to 6 using the \( \delta \)-reduction rule for ‘+’.

2.2 Representing and Evaluating a Program

Having seen how functions and expressions are represented, it is easy to think of a program as consisting of a series of unreduced \( \lambda \)-expressions that are evaluated, to a normal form using reduction rules. A naive scheme for the evaluation of a program could therefore be:

```
begin
  Starting from the top-level expression
  while redexes exist do
    select a redex
    reduce the redex
    overwrite the redex with the result
  end
end
```

Unfortunately, at any reduction step, there may be more than one redex available and having picked a redex it is possible that the series of reductions that result do not terminate.

Before explaining the choice of redex for reduction, the different types of redex need to be defined:

The *leftmost* redex is the redex that occurs textually to the left of all other redexes within the expression, and similarly for the *rightmost redex.*
An outermost redex is a redex that is not contained within any other redex.

An innermost redex is a redex which contains no other redex.

There are two possible reduction orders:

Applicative-order reduction (AOR) which reduces the leftmost innermost redex first. This reduction order will result in the evaluation of the argument expression before attempting to substitute a variable into it. As a result, AOR is the basis on which eager evaluation can be performed, but, in attempting to evaluate before performing substitution, non-termination may result.

Normal-order reduction (NOR) which reduces the leftmost outermost redex first. This reduction order delays the evaluation of any redexes within the argument expression until there is no alternative redex available, by substituting variables before attempting any reductions within the argument expression. NOR is the basis on which lazy evaluation can be performed and as such can be significantly less efficient than AOR reduction, but, if there is an possible selection of a redex that results in the termination of evaluation, NOR reduction will result in this selection — the Standardisation theorem.

It is possible to avoid performing α-reduction to resolve name clashes that can occur during β-reduction when the argument expression contains free variables which occur bound in the body of the applied abstraction by preventing β-reduction from being performed in the presence of free variables. To do so, expressions are reduced to a ‘restricted normal form’ known as weak head normal form (WHNF), which for an expression $E$ holds if [Field and Harrison, 1988]:

- $E$ is a constant
- $E$ is an expression of the form $\lambda x_n. E'$ for any $E'$
- $E$ is of the form $PE_1E_2...E_n$ for any constant function $P$ of arity $k > n$. This rule states, that a partially applied constant function is also WHNF.
2.2 Representing and Evaluating a Program

As a result of reduction to only WHNF, evaluation of an expression stops before the function body of an expression is entered, no free variables are therefore encountered.

The model of expression reduction presented by the λ calculus is that of string reduction where intermediate expressions forming a reduction sequence are represented symbolically and indication that multiple occurrences of the same identifier refer to the same expression is done by performing multiple copies of the expression. This copying can be very inefficient, for it can result in the multiple evaluation of the expression. It is possible to prevent these multiple evaluations by performing AOR, but, as has already been explained, this may result in non-termination. It is, of course, much more desirable to use NOR, but to do this the concept of sharing must be introduced that ‘links’ the copies of the argument expression, and results in all copies being updated with the (WHNF) value that results from the first evaluation of the expression.

Having presented reduction ordering, normal forms and sharing, it is now possible to define more fully eager and lazy evaluation schemes. Eager evaluation uses AOR to reduce expressions to WHNF and corresponds to the call-by-value mechanism of imperative languages. If however, NOR is used, either with α-reduction or reduction to WHNF for the avoidance of name clashes, the calling mechanism corresponds to call-by-name. However, as has already been demonstrated, the multiple evaluation of expressions is possible. The addition of sharing to call-by-name results in call-by-need, which in a functional language behaves equivalently, because it is not possible for an argument expression to perform side-effects. The final addition of lazy constructors, where arguments to constructors are assumed to be unevaluated, results in lazy evaluation.

There are two models of reduction and expression evaluation, environment-based and copy-based and over the following sections the most common abstract machines that implement these for execution of functional language programs are explained.
2.2.1 The SECD machine

The SECD machine [Landin, 1964] provides an environment-based mechanism, for the ‘mechanical evaluation of \( \lambda \)-expressions’ using four stacks, represented as a (stack) tuple, \((S, E, C, D)\) where:

- \( S \) — is the stack of objects used in recursive expression evaluation.
- \( E \) — is an environment.
- \( C \) — is the control string that is the remaining part of the expression currently being evaluated.
- \( D \) — is the dump, a previous state (partially) restored on return from a function call.

The machine executes using a state transition function to perform a series of state transitions when supplied with the current state (i.e. the current state of each stack represented by the stack tuple) and determines the next state using the control string \( C \).

At each step the possible machine operations are:

If \( C \) is empty, the current (sub-) expression has been fully evaluated and its result, \( R \), is the single element on the \( S \) stack. The \( D \) stack is popped and this top state restored followed by the pushing of \( R \) onto \( S \) (remember, that the \( S \) that \( R \) existed on is not now the current \( S \), for \( D \) has been popped).

IF \( C \) is non-empty and has object \( X \) at its head, then one of the following operations is performed:

- If \( X \) is a constant or an identifier, the value of \( X \) in the current environment is pushed onto \( S \) and \( X \) is popped from \( C \).
- If \( X \) is a \( \lambda \)-abstraction then a closure is constructed and pushed onto \( S \).
- If \( X \) is a function application where the function \( F \) is applied to its argument \( A \), then, \( X \) is popped from \( C \) and, for the eager evaluation implementation, \( F \) and \( A \) are evaluated and the values of \( A \) and \( F \) along with the apply symbol ‘@’ are pushed onto \( C \).
2.2 Representing and Evaluating a Program

- If $X$ is the @ symbol the basic operation is to apply the top element of $S$ to the second element. This operation is performed as is if the function at the head of $S$ is a primitive function. If however, it is a closure then the body of the expression within the closure must first be evaluated for its associated environment before the application may occur.

In order for the SECD machine to be altered to operate with a lazy evaluation scheme, the sharing of argument expressions must be allowed. This is achieved by assignment to the suspensions representing the unevaluated arguments. The following three modifications are required:

- Reconsidering $X$ as a function application: instead of evaluating $A$, a suspension that represents $A$ is created and pushed onto $S$ where its value would have gone in the eager scheme, the actual application is performed as before.

- When using the call-by-name mechanism, dumps must be specially tagged as either application-dumps or suspension-dumps, for when a primitive function is applied to a suspension, the suspension must be evaluated and then the re-application of the function performed, this therefore requires the saving of the current state on $D$. If however, the call-by-need mechanism is being used, suspensions on $S$ are represented by pointers to their actual locations, so the location of the suspension must also be saved before its evaluation. After evaluation, the resulting value is stored at that location so that the other references to the suspension can share the evaluated result — multiple evaluation is prevented!

- Reconsidering an empty $C$ and a suspension-dump as the current dump: When using the call-by-name mechanism the state saved in the dump is restored with the argument’s value that is at the head of $S$ and the suspension beneath this value on $S$ is replaced by the dump, the primitive function that forced the suspension will then be re-applied. For the call-by-need mechanism, the location saved in the dump is updated with the argument’s value that is at the head of $S$
and then the state in the dump is restored and the primitive function
re-applied.

2.2.2 Graph Reduction

An alternative technique to the environment (stack) based handling of \(\lambda\)-
expressions that forms the underlying principle of the SECD machine is that
of graph reduction [Wadsworth, 1971], where an expression is represented as
a directed graph (usually drawn as an inverted tree). Each node represents
a function call and its subtrees represent the arguments to that function.
Subtrees are replaced by the expansion or value of the expression they rep-
resent. This is repeated until the tree has been reduced to a value with no
more function calls (a normal form).

The above is a rather simplified explanation of graph reduction, in actual
fact, constants are represented at the leaves of the tree; function applica-
tions of the form \(FA\), the application of \(F\) to \(A\) are represented using an
apply-nodes;\(\@\), with \(F\) on the left-hand leaf and \(A\) on the right; curry-
ing is used to write multi-argument functions as a series of one argument
function applications to yield the binary pair that is needed for the tree rep-
resentation; finally, \(\lambda\)-abstractions of the form \(\lambda x. M\) are represented using a
\(\lambda\)-node \(\\lambda\), with the bound variable \(x\) on the left-hand leaf and the (subtree)
representing the body \(M\) on the right.

Consider an expression graph \(G\). The left spine of \(G\), corresponding
to the chain of application nodes along with the root, is found by taking
successive left branches from the root node. Evaluation of an expression
within \(G\), and hence reduction, begins by locating the next redex node in
the graph. For NOR all that must be done is to traverse (unwind) the left
spine of \(G\) until the node currently being traversed is no longer an apply-
node. The current node will be one of the following and the appropriate
actions are taken until the expression is in WHNF:

- Atomic data object — the expression is in WHNF and no further
  reductions are possible.

- Composite data object — the sub-graph 'hanging off' of the current
  node has a (primitive) constructor at its root. There are no reduction
rules for constructors so the expression is in WHNF and no further reductions are possible.

- **λ-abstraction** — if there are no sub-expressions hanging off of the current node, then there are no outer apply-nodes and so the expression is in WHNF, otherwise the sub-expressions must be reduced and therefore the next redex is the first apply-node that is found above the current node on the spine.

- **Primitive (constant) function** — defined to take \( k \) arguments, i.e. with \( \textit{arity} \ k \). If the function is being applied to less than \( k \) arguments, say \( n \), then the expression is a partial application of the primitive, which by definition of WHNF, is in WHNF and, as a result, the expression cannot be further reduced until it is applied to \( k - n \) more arguments. If \( k \leq n \) then the expression is reduced using the primitive’s \( \delta \)-rule.

Having located the redex for reduction, \textit{graph transformations} are performed that are meaning preserving and which correspond to sub-expression reduction. The result of \( \beta \)-reduction, performing the application of a \( \lambda \)-abstraction sub-graph to an argument sub-graph, is a variant of the \( \lambda \)-abstraction sub-graph where each leaf node representing the bound variable has been replaced with the argument sub-graph. To provide sharing and hence cater for lazy-evaluation, each pointer to a bound variable leaf within the function graph is replaced with a pointer to the root node of the argument sub-graph, the redex node is then overwritten with the result (the root of the modified function graph). Unlike the environment-based mechanism of the SECD machine, graph reduction is copy-based, the application reduction rules result in the copying of the body of the applied \( \lambda \)-abstraction so that the substitution of bound variable nodes occur within the copy itself. The copy of the body requires new graph nodes for every node other than its root node, which is written into the redex node as required. Having performed the argument substitution of \( \beta \)-\textit{reduction}, the redex is physically overwritten by the result of its reduction. This updating of the redex may result in other nodes in the redex becoming ‘disconnected’ from the root node and if these nodes are not shared then they are no longer required in
the evaluation and the space that is allocated to them can be reclaimed by the garbage collector, i.e. they are garbage cells. As indicated above, the rules for transformation of a graph corresponding to the application of a primitive function are given by the $\delta$-rules of the primitive. For a primitive function of arity $k$ [Field and Harrison, 1988]:

- Those arguments which are determined to be strict arguments of the primitive, (arguments that must be evaluated before the function can be applied), as specified by the $\delta$-rules, are evaluated by rewinding (moving back up) the spine and applying the graph reducer to the appropriate graphs.

- The primitive function application is performed using the $\delta$-rule and evaluated arguments.

- The redex node is overwritten (by rewinding back up the spine) with the resulting graph from the application. This, as above, may result in the creation of garbage cells which must be collected at some time by the garbage collector in order to prevent space leaks.

It is the evaluation of the top-level expression, performed by unwinding the spine of its graph to find the first redex and then reducing it, that results in the generation of further reductions, possibly involving recursive calls to the reducer. It should be a fairly intuitive deduction, that a stack can be used for the management that is needed to handle the unwinding of the spine of a (sub-)graph, initiation of reduction of an argument sub-graph, completing the reduction of the (sub-)graph and the rewinding of the spine.

The above provides a very brief overview of how a graph reducer evaluates expressions, by searching for the next redex to evaluate and then performing transformations until the redex is in WHNF, for a slightly extended $\lambda$-calculus that includes constants and primitive functions. It suffers an unfortunate inefficiency, namely that when a function application is performed, the function body must be copied before the binding of free variables to their values can occur. A combinator implementation, eliminates the need to perform the copy by transforming each $\lambda$-expression into an equivalent expression built only from applications of primitive functions
and combinators, each of which is a closed expression. This can be achieved through variable abstraction into a fixed set of combinators or by lambda lifting. Lambda lifting is the process of abstracting the free variables from one λ-body at a time and replacing the λ-abstraction by the partial application of a new combinator which has one more argument than the number of free variables abstracted, where the extra argument corresponds to the bound variable of the λ-abstraction. Further details of the reduction rules for such a system can be found in chapter 12 and 13 of [Field and Harrison, 1988].

2.2.3 The G-machine

Although the basic graph reduction model requires the copying of the function body, it has several advantages over the environment-based model, notably, that sharing is supported without the need for an environment structure and that NOR evaluation is easily expressed and relatively efficient to implement. These advantages suggest that graph reduction is a most suitable computational model for the implementation of lazy functional languages. It is for this reason that the G-machine [Johansson, 1984], a compiled lazy implementation for functional languages that results in an abstract machine was designed, and a variant of it has been adopted by GHC.

The Compilation Process

Before describing the G-machine, the following section outlines the phases of compilation from a high-level language through to a machine-dependent language, as well as explaining the rationale for a target language.

It is not practical to write programs in the λ-calculus just as it is not to specify them as Turing machines, as a result high-level languages are defined that are less restrictive and provide all the functionality that a programmer would expect such as named expressions, typed variables and structured types, global and local definitions, pattern matching and polymorphism, input/output and foreign language interfaces to name but a few. The result of this is that a program in a high-level language, of which Haskell is the example that is discussed here, must undergo several compilation phases that
result in a *target language* that is either interpreted or further compiled into a low-level language such as machine code or assembler.

![Compilation Route Diagram](image)

Figure 2.1: The compilation route — High-level language to target language.

Figure 2.1 shows the ‘compilation route’ that a high-level language program undergoes to reach the desired target language, the stages are as follows:

- **Parsing** — (in the above context, for it traditionally means just syntax analysis) consists of three phases:
  
  - *Lexical analysis* — The source code of the high-level program is traversed from left to right and grouped into *tokens* that are sequences of characters with a collective meaning which then become input to the following phase. The white-space characters that separate the tokens are eliminated.
  
  - *Syntax analysis* — The tokens from the previous phase are grouped into grammatical phrases that are used by the compiler to synthesise output. The grammatical phrases can be represented by a *parse tree* that describes the syntactic structure of the input. More usually, the internal representation adopted is that of an *abstract syntax tree* that describes the syntactic structure and is a compressed representation of the parse tree in which the operators appear as nodes and its operands as children of the operator node. Syntax analysis is the process of determining whether a string of tokens can be generated by a *context-free grammar* that defines the syntax of the language.
2.2 Representing and Evaluating a Program

- **Semantic analysis** — The semantic analysis phase checks the source for semantic errors and gathers type information for the subsequent code generation phase. It uses the hierarchical structure that is the abstract syntax tree to identify the operators and operands of expressions and statements. An important part of this phase is type checking where the compiler checks that each operator has operands that are permitted by the source language specification.

These three phases result in the construction of a symbol table that records attributes pertaining to the program identifiers such as type, storage allocated, scope etc. This structure allows the record for each identifier to be quickly retrieved, and subsequent phases use this information in different ways, for example, identifier scope and type for code generation.

- **Intermediate code generation (Simplification)** — After these three phases, some compilers generate an explicit intermediate representation of the source program that is a ‘syntactically sugared’ and somewhat expanded (as well as typed) variation of λ-calculus, using denotational semantics, i.e. this λ-calculus variant is used as a meta-language; the meaning of a program is expressed by mapping it into a corresponding λ-calculus object [Hankin, 1994]. In a ‘typical’ functional language compiler this stage can be further subdivided into two phases.

  - First intermediate form — The first phase acts on the abstract syntax tree to produce the first intermediate form through translation and the elimination of pattern matching.
  
  - Second intermediate form — The second phase produces the final intermediate language representation through lambda lifting and combinator generation.

- **Target language/abstract machine code generation** — It is theoretically possible (and sometimes performed in practice) to compile straight from the abstract syntax tree into a low-level machine-dependent lan-
Functional Language Implementation

guage. There are however several benefits of compiling to a target language or abstract machine code [Aho, Sethi, Ullman, 1986]:

- Re-targeting is facilitated; a compiler for a different machine can be created by attaching a back end (that which performs machine-dependent optimisations and/or language code generation) for the new machine to a front end of an existing one.

- A machine independent code optimiser can be applied to the abstract machine language representation.

This abstract machine representation should have two important properties; it should be easy to produce, and easy to translate. This representation must, in general do more than just compute expressions; it must also handle flow-of-control constructs and procedure calls. Examples of this type of code is that produced by a compiled (as opposed to interpreted) implementation of the SECD machine or the G-machine.

- **Code optimisation** — This stage attempts to improve the code by applying meaning preserving transformations to the target language code, so that faster running code results which takes up less space. Examples of optimisations are the unravelling of loops and in-lining of procedure calls.

- **Code generation** — The final stage is the generation of low-level language code normally consisting of relocatable machine code or assembly code. The issues involved in the generation of the code is machine dependent and usually involves operating system and memory management system calls, instruction selection, register allocation and evaluation order.

**Structure of the G-machine**

The G-machine, like the SECD machine performs execution steps through state transitions and a state is represented by a 7-tuple \((O,C,S,V,G,E,D)\) where:
2.2 Representing and Evaluating a Program

- $O$ — is the output produced so far, consisting of a sequence of integers and booleans.

- $C$ — is the $G$-code (the language of the G-machine) sequence currently being executed. When mapped to the target machine it is the target code of the currently executing function and the program counter.

- $S$ is a stack of pointers into the graph being evaluated. On the target machine, this maps to a data area for the pointer stack and a stack pointer register, $ep$.

- $V$ — a stack of basic values, integers and booleans on which the arithmetic and logical operations are performed. On the target machine, it is mapped to the system stack and stack pointer, $sp$.

- $G$ — the graph, a mapping from node names to nodes. On the target machine, this maps to the heap, divided into two equal halves and a register pointer, $hp$, that points to the next free location. It is this structure that is of the most significant interest where garbage collection is concerned, for it is storage area for structures generated during the evaluation of the program. When these structures are no longer used in evaluation, they become garbage cells and can be reclaimed by the garbage collector. New structures are added to the heap by storing them at the current free location indicated by $hp$. When the storage space in the heap is exhausted, the garbage collector is invoked to release the memory currently allocated to the garbage cells. Garbage collection techniques will be explained in some detail in chapter 3.

- $E$ — a global environment, mapping from function names to pairs consisting of the number of curried arguments and its code sequence. E corresponds to the code segment on a target machine.

- $D$ — is a dump used for recursive calls to the expression evaluation function. It is a stack of pairs consisting of a stack of node names, $S$, and a G-code sequence, $C$, before evaluation. On a target machine, this corresponds to the system stack and stack pointer, $sp$. Unlike in
the abstract machine, only pointers into the $S$ stack and system stack are pushed, not entire stack dumps.

The G-machine implementation uses four compilation schemes [Field and Harrison, 1988]:

- The $C$-scheme — which generates the code to construct graphs representing expressions.

- The $E$-scheme — which generates code to evaluate expressions represented by the graphs constructed from the code generated by the $C$-scheme, leaving a pointer to the result at the top of the evaluation stack, $S$.

- The $F$-scheme — which generates the function bodies of user-defined functions, reducing the graph constructed by the code from the $C$-scheme, to yield the value that results from the function application.

- The $B$-scheme — similar to the $C$-scheme, but compiles code to evaluate integer and boolean valued expressions leaving their value on the dump stack, $D$.

The execution of a compiled G-machine program is based on the computational model presented in 2.2.2. The $S$ stack is used to unwind and rewind the spine of a graph and together with $D$, to evaluate function applications. It is hoped that enough detail has been presented in the section 2.2.2 and section 2.2.1 to enable the reader to understand the basic operation of the G-machine without any further explanation.

This completes the overview of the concepts that underly the implementation of functional programming languages, and the following section now introduces GHC and the principles behind it, namely the Core language and Spineless Tagless G-machine.
2.3 Introducing *GHC*,
The Glasgow Haskell Compiler

2.3.1 The Goals of GHC

The ‘GHC Team’ state three goals as their reason for the writing of GHC [Peyton Jones et al., 1992]:

- ‘To make freely available a robust and portable compiler for Haskell that generates good quality code’.

- ‘To provide a modular foundation that other researchers can extend and develop.’ — They have tried to build their compiler in a ‘well-documented and modular way, so that others will find it (relatively) easy to modify.’ It is also their belief that ‘researchers are often unable to evaluate their ideas because the sheer effort of building the framework required is too great.’ As this report will demonstrate, such modification is not trivial, on the other hand neither is modification of any industrial (non-toy) compiler, but they have certainly achieved this goal in providing a framework that has allowed for the investigation of the ideas presented in [While and Field, 1992].

- ‘To learn what real programs do.’ — ‘Lazy functional programs execute in a particularly non-intuitive fashion’ and so they planned to ‘make careful, quantitative measurements of their behaviour’.

2.3.2 The Structure of the Compiler

The compiler consists of the following nine modules [Peyton Jones et al., 1992]:

- A ‘main’ and ‘simple’ recursive-descent parser that produces an abstract syntax tree that ‘faithfully represents every construct in the Haskell source language, even where the distinction is purely syntactic. This improves the readability of error messages from the type checker.’
A renamer that ‘resolves scoping and naming issues, especially those concerned with module imports and exports.’

A type inference module that ‘annotates the program with type information, and transforms out all the overloading.’

A desugarer ‘converts the rich Haskell abstract syntax into a very much simpler functional language called the Core language. Because desugaring follows type inference, type error messages are expressed in terms of the original source and appear more quickly.’

‘A variety of optional Core-language transformation passes improve the code.’

‘A simple pass then converts Core to the Shared Term Graph (STG) language, an even simpler but still purely functional, language.’

‘A variety of optional STG-language transformation passes improve the code.’

‘The code generator converts the STG language to Abstract C. Abstract C is no more than an internal data type that can be printed in C syntax.’

‘The target-code printer prints Abstract C in a form acceptable to a C compiler.’

The ‘key-idea’ behind the design decisions of GHC is that ‘most of the compilation process is expressed as correctness-preserving transformations of a purely functional program.’

What should be apparent is that this compiler design follows almost exactly, the compilation process that was explained in section 2.2.3. What was avoided in the previous sections was elaboration on an intermediate language and a more accurate definition of program representation and evaluation presented in section 2.2. The reason for this, was so that these concepts could be explained with reference to the Core- and STG- languages and the (abstract) Spineless Tagless G-machine.
2.3 Introducing GHC,
The Glasgow Haskell Compiler

2.3.3 The Spineless Tagless G-machine

The STG-machine [Peyton Jones, 1992], like the G-machine, is a supercombinator graph reduction implementation. This means, that it uses lambda lifting to abstract whole sub-expressions containing free variables as opposed to just the free variables themselves, the result is the elimination of $\lambda$-abstractions in favour of supercombinators and reduces the number of function applications which are ultimately required [Field and Harrison, 1988]. The STG-machine differs from the G-machine in three ways:

- The abstract machine code for a program consists of a sequence of abstract machine instructions, each of which has precisely defined operational semantics using state transitions. However, the STG-machine, has as its abstract machine language a 'very small functional language, which has the usual denotational semantics.' In addition to these denotational semantics each language construct has corresponding operational semantics for the same language which can be (and are!) expressed using a state transition system. Compilation rules are therefore defined for the conversion of a functional program into abstract machine code.

- The STG-machine is spineless — In the G-machine, as reduction proceeds, a stack is constructed consisting of pointers to the arguments pointed to by the spine. The observation that these arguments are always of the same form $(c, \text{tup})$, for some supercombinator $c$ and tuple $\text{tup}$ results in the stacking of the (root of the) arguments themselves. Each item on the spine stack is now a pair of a code pointer and a pointer to a tuple, which can be thought of as an application node, the code defining a function which is applied to the tuple. This code pointer/tuple pointer pair is called a closure, and a tuple of such closures is called a frame. A pointer to a frame is thus a frame pointer. As a result of this, there is no longer a spine in the heap; the stack is the spine of the expression being evaluated' [Peyton Jones and Lester, 1992].
- The STG-machine is *tagless* — Tagged implementations, such as the G-machine, require that objects are tagged so that (heap) objects built with different constructors can be distinguished from one another. The STG-machine eliminates tags through the uniform representation of its heap objects (as closures) by placing a code pointer in their first field. The object need not now be distinguished, because each object is handled by the code pointed to by this code pointer, so on evaluation, all that need be done is to jump to this code. This will be discussed further in section 5.2.

**The Core Language**

The definition of the Core language is at most sufficient to efficiently express the full range of Haskell programs, and no more. Programs are represented in the Core language by a series of supercombinator definitions, where 'series' implies the existence of at least one. In the functional language Haskell the main function represents a supercombinator from which the evaluation of the program is initiated. ‘Simple’ variables of a supercombinator are better explained using an example:

```plaintext
  double x = x * x
  main     = double 2
```

Both main and double are supercombinators and x is a simple variable.

When a supercombinator does not take any arguments, it is called a *constant applicative form* (CAF), of which, double2 is an example:

```plaintext
  double2 = double 2
```

The core language is an extended $\lambda$-calculus that contains the additional expressions: let, that allows local definitions within supercombinators, letrec that makes possible recursive definitions, and case that eliminates the complex implementation of pattern matching, whilst allowing Haskell to support it.
The definition of program evaluation presented earlier can now be refined:

begin

Starting from the top-level expression
while redexes exist do

Locate the outermost function application (although
it may not necessarily be reducible) by:

Unwinding the spine until a supercombinator
or primitive function is hit.

Performing an argument check on the number
of arguments that the function takes.

Rewinding the spine by that number of
application nodes; the redex is at the
node where the rewind terminates.

Reduce the redex.

Overwrite the redex with the result.
end
end
Chapter 3

Garbage Collection and Basic Collectors

Programming languages are increasingly being designed which alleviate the programmer from the chore of dynamic memory management, where memory, may be allocated and reclaimed for data whose lifetimes are not determined by lexical scope. It is therefore one of the tasks of the program’s runtime system to manage this dynamic memory and determine when it is no longer required and can subsequently be reclaimed. This is done using the part of the runtime system known as the garbage collector, which is normally invoked when the mutator has exhausted its free memory store. Hence garbage collection may be defined as ‘the automatic management of dynamically allocated storage’ [Jones and Lins, 1996].

The following sections give a brief outline of the different algorithms of which today’s collectors are variations.

3.1 The Reference Counting Algorithm

For each object being managed by the collector, a count is maintained of the number of pointers referencing it from other objects (including itself). When a pointer is assigned to the object (i.e. set to point to it) the reference count is incremented and when this reference is destroyed, for example, by assignment to ‘null’ or to another object, then the count for that object is
decremented. If the reference count is equal to zero then the object is no longer 'live' (being used) and so can be reclaimed by the garbage collector and the memory allocated to it returned to the 'free pool', the area of memory from which objects are allocated their storage space. However, before the collector can reclaim the memory allocated to the cell, any cells which are referenced by this garbage cell must also have their reference counts decremented and as a result other cells may also become garbage. The collector therefore 'chases' references iteratively or recursively applying the algorithm to those cells reachable via the reference chain.

The main advantage of this scheme is that the memory management overheads are distributed throughout the computation of the user program, with the reference count handling and the reclamation of garbage cells interleaved with its execution. This is in contrast to 'stop-start' algorithms which result in the pause of the user program, while the collector runs. Incremental collectors are preferable in many applications, in particular real-time systems. Another advantage is that it is a relatively simple scheme to implement, usually by defining a macro to handle the reference counter when pointer assignment occurs.

Unfortunately, there are more disadvantages to the scheme than there are benefits and so it is a little used scheme. The distribution of processing (i.e. the granularity of interleaved processing) is rather 'lumpy' with the cost of deleting the last pointer to a sub-graph depending on its size [Jones and Lins, 1996]. Also, circular structures cannot be collected easily.

The efficiency of the scheme as a whole is very low with a high price being paid for the simplicity of the scheme - a high processing cost in updating the counters that maintain the reference count invariant. Each time a pointer is updated, the reference counts in both the old and possibly the new target objects have to be adjusted. Finally, the most serious disadvantage in the scheme is in its inability to reclaim cyclic structures, (which incidentally, occur frequently in lazy functional languages for the handling of such concepts as recursion) which may result in serious space leaks.
3.1.1 Lazy Garbage Collection

A scheme that can be used to solve the problem of 'lumpy' processing suffered by the standard reference counting algorithm and that reduces the (serious) delays in the user process whilst the garbage cells are reclaimed is that of lazy garbage collection [Glaser and Thompson, 1985].

A 'to be decremented stack' (TBD stack) is used which contains the addresses of those cells whose reference count is to be decremented. The reference count of a cell is incremented as before, but when decrementing the count, references are no longer chased along the reference chains, but the address of the cell that is to be decremented is pushed onto the TBD stack.

The core of the algorithm is executed on allocation of a new cell. The algorithm iterates over the TBD stack, popping the cell address on the top of the stack. If the reference count of the cell is one, then the cell is garbage, for it is due to have its reference count decremented to 0. Hence, the new cell can be allocated using the memory for this old cell once the addresses of any cells referenced by this garbage cell have been pushed onto the TBD stack for decrementing. Of course, this applies for the allocation of fixed sized cells, but is equally applicable for variable sized cells. If the memory released by the deallocation of the garbage cell is insufficient to fulfil allocation of the new cell then the next cell address on the TBD stack is popped. If the reference count of a cell is found to be greater than one then the cell is clearly not garbage and so the count is simply decremented. The algorithm (successfully) terminates when enough memory has been 'collected' so as to allow for the allocation of the new cell from the TBD stack, or the TBD stack is exhausted and the cell is allocated using the free list as before. Obviously, the mutator will terminate indicating that it has been unsuccessful in allocating the new cell if the collector was unable to allocate memory from the TBD stack or from the free list.

Whilst garbage collection now occurs lazily, with the majority of the overheads arising when a cell is allocated as opposed to when references are created and destroyed, it is still conceivable for mutator delays to occur. If there are a large number of cell addresses, whose reference counts are
greater than one, at the head of the TBD stack, then the length of the delay will depend on how long it takes the collector to traverse the stack decrementing the reference count of these cells until it finds the address of a cell whose count is one. Fortunately, it has been shown that most cells in a functional program exhibit a reference count of one [Stoye et al., 1984] and so it is unlikely that the collector will have to make a prolonged traversal of the TBD stack before finding a garbage cell. As such, the scheme is (demonstrably) more efficient than the chasing of references down each reachable reference chain as performed by the standard algorithm.

Field and Harrison observed that 'it is still possible, in pathological cases, for there to be a long chain of shared cells referenced at the top of the TBD stack' [Field and Harrison, 1988]. They resolve the problem, improving the real-time performance of the collector, by combining the TBD stack with a conventional push-only heap. A limit is set on the number of addresses that the collector can pop off of the TBD stack without finding a garbage cell. If this limit is hit, then the cell is allocated using heap allocated space by pushing the cell onto the heap. If the push-only heap space is exhausted then there is no other option than to continue popping cell addresses off of the TBD stack; if both the TBD stack and push-only heap are exhausted, then the mutator is forced to abort.

3.1.2 Single-bit Reference Counting

The observation of Stoye and his associates is capitalised on in an algorithm that uses just a single bit to record the reference count instead of a whole word. With this reference count bit set to zero, the count indicates just a single reference to the cell, whilst a count of one indicates that two or more references exist. When a reference to a cell is first created, the count bit is set to zero, when a second reference is created, the count bit is set to one and for subsequent references nothing need be done. When the garbage collector is invoked, if a cell is found whose reference count bit is set to zero, then the cell is garbage and can be collected; the reference chain from this garbage cell can then followed to determine if any cells, whose 'life' was being maintained by the garbage cell, can now be collected. If however,
3.1 The Reference Counting Algorithm

The reference count bit of the cell is set to one, then the count cannot be decremented to zero, for all that is known is that there are at least two references not only two. It is for this reason, that a second type of garbage collector is needed that is invoked when the reference count bits of all the cells have been set to one. Fortunately, because the majority of cells are found to have a ‘true’ reference count of one (i.e. the majority of cells will have their count bit set to zero in this scheme) this second collector will be invoked much less frequently than if employed on its own.

The algorithm can be optimised by placing the reference count bit within the reference pointer itself. When a reference to a cell is first created, the bit is zero, as before, but on creation of a second reference, the count bit must be set to one in both this new reference pointer and the original pointer; for subsequent references, only their own count bit need be set to one. The principle for collection is the same as before, except it is the pointer and not the cell that is being examined (for it contains the count bit). The advantage to this scheme is that when references to a cell are created, the cell itself need not be referenced and since the reference is being accessed anyway, examining and setting the count bit requires little extra work (none in terms of memory accesses); the same also applies when destroying the references, access to the pointer occurs anyway in the dereferencing. As a result, there is a saving of one memory access per pointer copy in a sequential instruction system and one (possibly remote) communication with the referenced cell in a distributed environment [Field and Harrison, 1988].

3.1.3 Weighted Reference Counting

The efficiency that the single-bit reference counting algorithm provides has meant that a variant of the scheme has been found to be particularly well suited to distributed implementations of functional languages where communication is so often the limiting factor in performance’ [Field and Harrison, 1988], the scheme is also currently used in the JAVA ORB.

This variant is known as ‘Weighted reference counting’ and is an extension of the single-bit reference counting algorithm that allows the representation of a reference count whose value is greater than one. The scheme was
developed independently in [Thomas, 1981] and [Bevan, 1985].

In this scheme, not only does the cell carry a reference count but reference pointers also carry a weight. Throughout the execution of the algorithm, an invariant is maintained such that the sum of the weights of all the reference pointers to a cell always equals the reference count contained within the cell. When a new cell is allocated, its reference count is set to a predetermined maximum value, and the reference count weight of the pointer to the cell is set to this same value. When subsequent references are created (i.e. the pointer is duplicated) the count weight of the pointer is equally subdivided between the new references, no access is made to the referenced cell. When a reference is deleted, the weight of that pointer is subtracted from the reference count contained in the cell (...so that the invariant is maintained). If, as a result of this subtraction, the reference count in the cell drops to zero, the cell becomes garbage and can be de/re-allocated (the references from the garbage cell are followed as before to seek out further garbage cells).

3.2 Mark - Sweep Garbage Collection

Mark-sweep algorithms exhibit several advantages over reference counting, the most significant of which are that cyclic structures/references pose no problem to the algorithm and there are no overheads associated with pointer manipulations. These benefits have lead to the implementation of (variants) of the scheme in some systems, such as the functional language Miranda.

In reference counting algorithms, the allocation space for a single cell can be recycled as soon as the last reference to it is destroyed. However, in a mark-sweep algorithm the garbage collector runs less frequently and is invoked when the cell allocation space, the heap, is exhausted. The garbage collector starts at the ‘root set’, the data that is immediately available to a program, without following any pointers (usually global, static, current local variables and register values). The collector then follows reference pointers from each root object to all the reachable objects, marking each one ‘found’ as it progresses. Once all reachable objects have been marked as found, all objects left unmarked, are no longer referenced by any other objects and
hence are now garbage and may be reclaimed by the collector.

The disadvantages of this scheme are that it is a stop-start algorithm and it also has a tendency to fragment memory, with objects being scattered across the heap. In later algorithms, it will be seen that memory fragmentation, which may render parts of memory unusable for considerable periods of time (because the available fragment is too small to fulfil the allocation of an object) may be eliminated through the compaction of the live objects in the system or by copying them to an empty allocation area where the live objects may be contiguously aligned. As I have already mentioned, the reference counting algorithm allows for memory to be recycled immediately whereas in this scheme, the mutator pauses its execution, the collector runs and then when collection is complete the mutator is permitted to resume. As a result of this stop-start execution, the cost, that is the mutator pause time, is dependent on the size of the heap since the collector must fully traverse it during the sweep phase. Whilst it is true that the reference counting algorithm suffers delays when traversing the reference chains, it is highly unlikely even in the worst case, that these chains contain all the cells within a functional environment (c.f. observation of Stoye et al.).

Many different variants of this algorithm have been developed to reduce the mutator pause time, the significant ones attempt to perform the sweep phase lazily by transferring the cost of the sweep to the allocation of cells [Hughes, 1982], [Zorn, 1989]. The pause is reduced (but more correctly spread) by performing the sweep phase in parallel with the mutator execution. This is not fraught with the problems that truly incremental algorithms (see 3.6) suffer, where the mutator can interfere with the garbage collector's progress (and vice versa), for the mark-bits of live cells are invisible to the mutator, and the collector will only modify fields accessible to the mutator in cells which are garbage (and by definition the mutator no longer has references to). On a single processor system, parallelism is simulated by placing an upper bound on the amount of sweeping that is done by the collector when a cell allocation occurs.
3.3 Mark - Compact Garbage Collection

The problem of fragmentation that the mark - sweep scheme exhibits may mean that it is impossible to place a larger object into the heap because none of the holes left are large enough to accommodate the new object even though the total amount of free space is sufficient. For smaller objects, the problem posed is what kind of algorithm to use when placing the new object in a hole. Finally, spatial locality exhibited by the objects in the heap may be destroyed, considerably reducing the efficiency of the mutator.

In order to solve the above problem, algorithms which compact the heap, eliminating the holes resulting from fragmentation, and leaving both the heap and the free-space store contiguous as opposed to ‘interleaved’, are used.

Such algorithms often use multiple passes to perform the compaction and work by calculating a forwarding address for the object under relocation. This forwarding address is stored in the object (for relocation in the next pass) or in the place of the object. The price paid for compaction and a contiguous heap is high and hybrid collectors have therefore been developed which use heuristics to determine when compaction needs to be employed [Sansom, 1991].

These algorithms are often employed when there is only a limited address space or the heap consists of a number of objects that have a consistently long life, hence the use of a ‘semi-space’ (copying) algorithm is inefficient and undesirable due either to space limitations or repeated copying.

Compaction of live data resolves the problems posed at the beginning of this section. Firstly memory allocation is simple, it is done by incrementing a heap pointer, and allocated in linear increments allowing variable-sized objects to be allocated with the same cost as other objects. Secondly, compacting live objects into a contiguous portion of the heap preserves and possibly increases the spatial locality of mutator objects.

There are three classifications of compacting algorithms [Jones and Lins, 1996], arbitrary, linearising and sliding. Arbitrary compaction algorithms move cells with no regard to their original order and whilst these algorithms can be simple to implement and fast to execute, it is possible for
orderings of cells to occur such that the resulting structures exhibit poor spatial locality through a reduction in cache hits and virtual memory performance. Linearising algorithms, where possible, move those cells which reference one another into contiguous heap locations (or at least as close to each other as possible), this benefits the spatial locality as mentioned above. Sliding algorithms shift the live cells to one end of the heap, eliminating free space fragments (free space will be contiguous from where the live cells end), preserving the order of the cells and thus the spatial locality.

One of the most well known compaction algorithms is presented in [Morris, 1978], however a more general algorithm exists, Jonkers' compaction algorithm [Jonkers, 1979], that imposes fewer restrictions on the organisation of the heap and yet works in a similar way by threading the live cells onto a list of those cells that point to it prior to invocation of the collector. When the cell is moved, the list is traversed and the pointers adjusted.

3.3.1 Pointer Threading

Threading is achieved using the technique of pointer reversal. Consider two locations $A$ and $P$ with $A$ pointing to $P$ and $P$ containing ‘Data’, threading is performed by reversing the pointer between $A$ and $P$ and storing the ‘Data’ contents of $P$ in $A$. For more than two locations, the following diagrams aid in explanation.

![Figure 3.1: Pre-threaded structure.](image)

Consider locations $A$, $B$, $C$ pointing to location $P$ (see Figure 3.1).
In order to thread this structure a list is created containing those locations that point to \( P \), with \( P \) at the head of the list, its original 'Data' contents stored in the tail and the next location reachable from \( P \) (see Figure 3.2). When \( P \)'s new location is established, the list is traversed and each pointer field updated with this value to restore the references to \( P \).

### 3.3.2 Jonkers' Compaction Algorithm

The collector is invoked as before, and marks the active structure. Two subsequent sweep passes of the heap are now required, the first threads and updates forward pointing references to \( P \) (\( \forall P \)), while the second threads those that point backwards to \( P \) (\( \forall P \)) and moves the cells. Consider just a single node \( P \), in the first pass the roots of the computation are threaded so that its references can be updated should any of the cells to which it refers be moved. The sweep continues, threading each pointer it encounters and updating the free space variable (the amount by which it is updated depends on the size of the cell, for fixed size cells, this increment is constant). On reaching \( P \), all forward references have been threaded and the 'Data' contents of \( P \) will have been placed on the end of the chain, the forward pointers can now be updated with the new address of \( P \), which is simply the value in the free space variable. Beyond \( P \), the sweep continues, threading pointers pointing back to \( P \). The second pass now begins traversing the heap, on reaching \( P \), the back pointing references are updated with \( P \)'s
new location by following its thread, (the new location is calculated in the same way that the new location for $P$ was calculated for the forward pointing reference updates), on completion of the pointer updates, the ‘Data’ contents of $P$ can be moved to its new location.

Although Jonkers’ algorithm is less restrictive than Morris’, it still requires three be enforced [Jones and Lins, 1996]:

- Pointers may only point to the header of a cell.
- The header must be large enough to contain an address.
- Headers must contain values that are distinguishable from pointers into the heap (although pointers to other areas of memory are possible).

However, Jonkers’ algorithm eliminated the restrictions placed on the direction of pointers; the need for the second pass to be in the opposite direction to the first; and the need for additional tag bits to distinguish ordinary pointers, threaded pointers and data.

### 3.4 Copying Garbage Collection

Garbage collectors that operate using a copying algorithm usually divide the free memory store into a number of partitions, generally two - on which we will focus. The first partition is adopted as the heap from which new objects are allocated. As the first partition becomes full and a new object is requested, a ‘flip’ occurs where the second partition now becomes the heap. The new object may now be allocated and all live objects from the old heap are copied into this second partition. The old heap partition now contains no useful data and will not be accessed until the next flip.

This is the most simple copying scheme, its efficient iterative implementation is attributed to Cheyney [Cheyney, 1970], and is an example of a ‘stop and copy’ algorithm where the mutator is paused while the collector runs. The cost of copying is proportional to the volume of live objects rather than the size of the entire heap. The live objects are compacted, the benefits of which have already been explained. Efficient implementation of such
an algorithm is relatively simple in comparison to other forms of tracing collection.

The main disadvantage of a copying scheme is that the size of the ‘usable’ heap is halved. However, on a virtual memory system, the ‘old’ partition may be swapped to disk. There are however performance implications associated with this, for example, the costs of paging. If the lifetime of objects is long then the costs of copying the objects at each flip is expensive and can result in a long delay before the resumption of the mutator. In functional languages however, it is accepted that low survival rates are typical and thus lifetimes are short [Peyton Jones et al., 1992].

### 3.5 Generational Garbage Collection

Generational garbage collectors are implemented in environments on the premise that most objects ‘die young’. As a copying variant algorithm, the heap is divided into a number of ordered partitions known as generations, where the ‘youngest’ houses newly allocated objects. Objects are promoted, at first from the youngest generation, and then subsequent generations, as they age. What differentiates one generational garbage collector from another usually depends on the number of generations being handled, the criteria used to determine the age of the objects, the frequency at which differing generations are collected and the intra-generation collection algorithm employed.

One of the most basic and obvious criteria to use in determining object age is its *wall-clock time* - the age of the object, in time, since its creation. This is however machine-dependent and it has been suggested that a better measure may be the number of bytes of heap space allocated [Jones and Lins, 1996], or the number of collection cycles that the object has lived for.

Once an object reaches a threshold age determined by the above criteria, it is promoted into an older generation.

When a generation becomes full, the collector is invoked. Using the root objects in the older generations and pointers to younger generation objects, it collects objects no longer in use. A collection policy is employed that determines the frequency at which the generations are collected. Generally,
the younger partitions are collected more frequently than the older ones. The collection algorithms used are implementation dependent and it can be quite beneficial to use different algorithms to collect different generations, so that the dynamic properties of the specific generation being collected are exploited [Sansom, 1991].

Because it is possible to collect a generation independently of other generations the mutator pause time is less than that of a collector that pauses for a full collection since fewer objects are traced and copied and the quantity of data moved over the entire program execution is reduced.

The disadvantages of such a collector arise in its implementation - determining the root objects within a generation is more difficult than determining the roots of the entire heap. Not only must the registers and stack be scanned but inter-generational pointers must also be found and treated as roots. These pointers are created either by the promotion of an object with pointer references into an older generation, or by storing in an object a pointer to an object in another generation. Two overheads are associated with the introduction of inter-generational pointers, the first is that the collector must keep track of them when promoting objects and the second is the need for a 'write-barrier' to trap pointer assignments by the mutator at time of storage. The use of a write-barrier for every pointer store would be extremely expensive, but the cost is substantially reduced as local variable pointer stores need not be trapped since they already belong to the root set.

Generational garbage collection was present in version 2.0x of GHC [Sansom and Peyton Jones, 1993] and the implementation was shown to outperform the one and two-space algorithm implementations. It was found that such a collector is extremely suited to lazy functional languages because objects do indeed exhibit the tendency to die young.
3.6 Concurrent and Incremental Garbage Collection

Although generational garbage collection is shown to reduce the mutator pause time, by concentrating reclamation efforts on a partition of the heap in which memory is most likely to be stored, it is not ideally suited to real-time or interactive applications. Real-time applications have performance criteria which must be met within specific time limits whilst in an interactive system, initiation of computation in response to interaction should appear real-time; the occurrence of an unbounded collection may violate these. Rather than attempting to improve the expected pause time at the expense of the worst case, concurrent collection demands guarantees for the worst-case performance. In reality however, a concurrent garbage collector may not meet its worst-case deadline but at best offers average-case pause times that are bounded by a small constant [Jones and Lins, 1996]. Many concurrent garbage collection algorithms were designed for execution on multiple processors. Here concurrent, implying parallel execution of both mutator and collector is distinguished from incremental, where the appearance of concurrency is created by interleaved execution of the mutator and collector processes on a single processor.

In a truly concurrent collector, garbage collection cycles occur in parallel with mutator advancement and unlike on a sequential system, a collection is often initiated using heuristics that aim to prevent allocation of a new object from ever failing, as opposed to when an allocation has failed. This is not however the case for incremental algorithms, collector initiation occurs at allocation failure and the collector attempts to perform only enough work (typically measured by the number of objects, \( k \) that are relocated) on each 'context switch'\(^1\) so that the allocation request may be fulfilled whilst progress is made. Care must be taken to prevent the mutator from exhausting memory before the garbage collection cycle is over.

To ensure this \( k \) is dynamically determined when collection is initiated. The following heuristic is widely used to determine the additional headroom

\(^1\)The term context switch is loosely used to describe the process of either mutator execution pausing and collector execution resuming or vice versa.
in the heap. It is worth noting that non-incremental (i.e. concurrent) collectors do not incur this overhead of additional headroom because allocation failure is unlikely to occur.

If there are \( R \) live words in the heap at the time of allocation failure, \( k \) words are evacuated at each new allocation and an average of \( C \) words are allocated for each new object, then \( R \) words will have been evacuated after \( \frac{R}{k} \) calls to the collector. Therefore, at the end of the collection cycle the number of live words will in the (new) heap is given by Equation 3.1.

\[
R(1 + \frac{C}{k})
\]

Therefore, in order to prevent starvation, the heap size, \( H \), must be greater than the number of live words given by this equation. Obviously, these amounts are doubled for a two-space copying collector. If the heap size is fixed, then Equation 3.1 can be rearranged so that the value of \( k \) can be dynamically determined for a particular collection cycle:

\[
R(1 + \frac{C}{k}) < H
\]

\[
k > \frac{RC}{H - R}
\]

An actual implementation of the collector may choose the units of \( k \) to be the number of objects scavenged or evacuated within the incremental cycle as opposed to the number of words evacuated/scavenged in the cycle. Clearly, by dividing the right hand side of Equation 3.3 by the average cell size, \( \langle C \rangle \), this can be achieved.

In both concurrent and incremental schemes, the mutator and collector can be perceived as being asynchronous processes with a shared address space. It is for this reason that a synchronisation barrier is needed to maintain data consistency to prevent such problems as mutator activity interfering with the collector’s recording traversal [Jones and Lins, 1996].

Suppose a collection cycle is in progress and that the collector has visited some objects and their direct descendants. If the mutator now writes a pointer into one of the objects that references an unvisited object it is pos-
sible that this unvisited and live object is reclaimed by the collector because it does not detect this pointer reference.

Generally, there are two types of barrier used, a read-barrier and a write-barrier. A read-barrier is used to prevent the mutator from accessing an unvisited object. When the mutator attempts the access, the mutator is blocked and the collector visits the object before allowing the mutator to continue. An alternative method is the use of a write-barrier. When the mutator writes the pointer references from visited to unvisited nodes, the write is recorded so that the collector can visit the object.

There are many different ‘flavours’ of read and write barriers and it is the choice of these along with their implementation that determines the efficiency of the collector. Both the While and Field collector and the algorithm proposed in this paper are variants of Baker’s incremental algorithm that uses a software read-barrier. Software read-barriers are accepted as being very expensive, suggestions that pointer loads may account for between 13 and 15 percent of a program’s instructions have been made [Zorn, 1990]. It is the intention of the While and Field collector and the variation presented here to eliminate the read barrier and its associated cost.
Chapter 4

Baker’s Algorithm

Because Baker’s algorithm is the motivating force behind the While and Field collector and the collector presented here, this chapter discusses it in some detail.

4.1 Algorithm Description

Baker’s classic algorithm is an incremental semi-space copying algorithm that uses a read-barrier to trap mutator access to objects which have yet to be copied from the exhausted heap space to the new heap.

The total memory store is divided into two partitions, Fromspace and Tospace. Fromspace is the exhausted heap space where both live and dead objects reside before initiation of the garbage collector and from which the collector will copy the live objects. Tospace is the new heap space where live objects are copied to, and from which new objects are allocated.

Like a stop and copy collector, the collector is initiated when the allocation of a new object fails due to the free space of the current heap, Tospace, being insufficient/exhausted. The mutator is paused and the algorithm begins by performing a flip of the two partitions with Fromspace becoming the new Tospace and Tospace becoming the new Fromspace. It is worth noting that this is not necessarily the best time to initiate the collector. Although the number of objects being copied is minimised by allowing the objects as much time as possible to die, it maximises the amount of heap allocated
and hence the number of page faults incurred [Jones and Lins, 1996]. Alternatively, spatial locality can possibly be increased and the number of page faults reduced (data compaction is maximised) by initiating the collector and performing the flip as soon as the previous collection is complete.

Unlike Cheyney’s stop and copy collector [Cheyney, 1970], where the collector would now be run to completion, the whole root set is evacuated from Fromspace into Tospaces, but only \( k \) objects are scavenged before the new object is allocated. Evacuation of an object entails copying the object to Tospaces, placing a reference to the copy on the scavenge queue and finally overwriting the Fromspace object with a forwarding pointer referencing the Tospaces object. Scavenging an object is performed by removing the object from the head of the scavenge queue and evacuating any (child) objects that it references. The garbage collector now pauses and the mutator resumes so that progress can be made. The next time that a new object is instantiated, the mutator is again paused, and the memory for the new object is allocated from Tospaces and another \( k \) objects scavenged before mutator resumption. In this way, garbage collection and mutator execution appear to be performed in parallel, with mutator pause times being reduced, and garbage collection being performed incrementally.

Because the collection is being done incrementally it is possible for the mutator to resume with the heap in an inconsistent state, where objects which have not yet been evacuated from Fromspace may be accessed by the mutator via references from pre-scavenged objects now residing in Tospaces. In order to prevent this, and provide the mutator with the illusion that garbage collection is complete at the end of each collection cycle (after \( k \) pieces of work), a read-barrier is used, to enforce the invariant of the algorithm that states that the mutator must never be allowed ever to get a handle on a Fromspace reference.

It is only possible for the mutator to access these pre-evacuated objects in Fromspace by issuing a pointer load operation. The read-barrier is therefore implemented by modifying the pointer load operation by performing a conditional test on the address of the object to see if it is residing in Fromspace and either evacuating it or following a forwarding pointer.

The diagrams below demonstrate the cycles needed for a complete Baker
garbage collection, and that the algorithm handles cyclic references. The
dashed lines represent the forwarding pointers, the dotted line separates the
two semi-spaces, and the shaded boxes indicate those objects currently on
the scavenger queue:

\[O_8\] \[O_9\] \[O_{10}\] \[O_{11}\] \[O_1\] \[O_3\] \[O_4\] \[O_5\] \[O_6\] \[O_2\] \[O_7\]

Figure 4.1: Heap at initiation of garbage collection: root set = \{O_1, O_2\}

\[O_8\] \[O_9\] \[O_{10}\] \[O_{11}\] \[O_1\] \[O_3\] \[O_4\] \[O_5\] \[O_6\] \[O_2\] \[O_7\]

\[O_1\] \[O_2\]

Figure 4.2: Semi-spaces after root evacuation.
Figure 4.3: Semi-spaces after one call to \texttt{mutator\_create()} and \( k = 4 \) scavenges.

Figure 4.4: Semi-spaces after two calls to \texttt{mutator\_create()}. Garbage collection completes before \( 2k = 8 \) scavenges. The memory allocated to the Fromspace objects may now be freed.
4.2 Pseudo-code Algorithm

The pseudo-code below, using a C style syntax, outlines Baker's algorithm and loosely demonstrates how a macro may be used to implement the read-barrier on pointer loads:

```c
/* Define the macro that implements the read-barrier on pointer loads */
#define mutator_pointer_load( pointer )\
   if ( pointer address is in from_space )\
      return pointer' = evacuate( object to which pointer points )\n   else return pointer

/* Routine called when the mutator requests the creation
 * of a new heap object
 */
mutator_create( object )
{
   if ( not enough space )
   {
      if ( Garbage Collector is running )
      {
         abort( out of memory error );
      }
      else
      {
         /* Mutator computation is now paused
          * while collector runs
          */
         start_Garbage_Collector();
      }
   }

   if ( Garbage Collector is running )
   {
      /* Do k pieces of work at each allocation */
```
for ( scavenged_count = 0; scavenged_count < k; scavenged_count++ )
{
    /* Remove and get next object to scavenged */
    object_to_scavenge = head( object_scavenge_queue )
    if( object_to_scavenge == NULL ) stop_Garbage_Collector();
    else scavenge(object_to_scavenge );
}
return allocate_memory( object );

/* Mutator execution now resumes */
}

start_Garbage_Collector()
{
    /* from_space becomes to_space and to_space becomes from_space */
    flip();

    /* For all roots, copy to new to_space */
    for ( root_count = 0; root_count < total_number_of_roots; 
         root_count++
    )
    {
        root = get_root( root_count );
        evacuate( root );
    }
}

evacuate( object )
{
    /* Copy the object to to_space */
    object_copy = copy_into_to_space( object );
    leave_forwarding_pointer( object, object_copy );
    scavenge_queue_add( object_copy );
4.3 Performance Implications

The above presents a generalisation of Baker’s algorithm. Baker includes two more considerations in his collection algorithm. If the root set is large, for example by inclusion of the program stack, it will not be possible to maintain a small upper bound when $\text{mutator}\_\text{create}()$ is called and a flip occurs (the entire root set is evacuated and $k$ objects scavenged). He therefore proposes that at each allocation request, $\text{mutator}\_\text{create}()$, $k'$ stack objects are also scavenged and at each flip $k'$ is recalculated so as to keep $k' \over k$ equal to the ratio of stack locations to heap objects. The second consideration results from the realisation that the cost of evacuating an object is relative to its size. He therefore chooses to evacuate large objects lazily. Not only does a Fromspace object contain a forwarding pointer but the Tosp ace object contains a pointer to the original Fromspace object. When a large object is evacuated, memory is allocated in Tosp ace for it and the forwarding pointer and backward link pointer are set. The rest of the Tosp ace object is now filled in incrementally, with the backward pointer being set to null on completion [Jones and Lins, 1996].

The overheads of the scheme are as follows:
• As previously stated, the cost of the read-barrier is very expensive, not only must a test be performed to find out if a garbage collection cycle is in progress, but if it is, a second test must be performed to see if the current object is in Fromspace or Tospace.

• The above considerations introduce further overheads:

  – Incremental scavenging of the stack complicates stack access routines (pops) and the read barrier must be applied to stack objects as well as heap objects.

  – Secondly, the lazy collection of large objects, specifically the use of the backward link pointer, requires an additional word and also a conditional test on the address of the object when a write access operation occurs. The test is used in order to determine whether the old object is used via the backward link or whether the new object should be used.

• The two final overheads, common to all incremental schemes, are:

  – The maintenance of a counter that records how many objects have been scavenged in the current incremental cycle, which is used to determine when the mutator should be resumed.

  – At allocation of a new object, a test must be performed to find out if a garbage collection cycle is in progress, for if it is, the collector will perform $k$ scavenges of the scavenging queue before allocating the object and resuming the mutator.
Chapter 5

GHC Runtime System and Heap Objects

5.1 An Overview of the New Runtime System

In section 2.2.3 the stages that a high-level program must go through in order to be compiled were described. An interpreter such as Hugs provides an environment for the management of the program and the interaction that it requires with the operating system and input/output devices. In addition to this, it provides a garbage collector to retrieve unreachable cells.

This section provides a brief overview of the new GHC runtime system (rts), it makes use of figures (based on) and extracts from The New GHC/Hugs Runtime System [Marlow and Peyton Jones, 1998] which should be referred to for further details.

Figure 5.1 shows the constituent components of the runtime system.

5.1.1 The Storage Manager

The Storage Manager consists of four components, spread across two layers. The first is a low level machine dependent layer known as the 'Block Allocator'. It provides an interface for the upper levels of the Storage Manager and runtime system so that they may demand the allocation or freeing of blocks of memory. The block allocator maintains a pool of free blocks and requests memory from the operating system as this pool is exhausted. A block is a
Figure 5.1: The components of the GHC runtime system.
power-of-2 sized chunk of memory which forms a group when contiguously allocated with other blocks. Memory is requested from the operating system in units of megablocks, an integer multiple of the block size, by doing this, these rts to OS requests are minimised; they are undesirable because the penalty for a request and a subsequent allocation is costly. Currently, blocks are 4kbytes and megablocks are 1Mbyte.

Each block has a block descriptor associated with it, which contains information pertaining to the block. A block descriptor stores:

- The address of the block
- The first byte of free memory in the block
- For the first block in the group, the number of blocks in the group
- For the generational garbage collector an identifier as to which generation/step the block belongs to. For a copying collector, an identifier indicating which semi-space the block is in - Tospace or Fromspace
- A link to another block descriptor

The link field is used to join both blocks and groups together to form a chain. Chains are used for the maintenance of memory allocated in blocks/groups to Fromspace, Tospace and the large_object_list.

There are several advantages in using a block-structured storage allocator, although it makes the implementation of a copying collector slightly more complicated. The major advantage is that areas of memory used by the system need not be contiguous, for example when making a subsequent request for allocation of further megablocks from the OS, there is no restriction that these megablocks be contiguous with previous megablock allocations. It is also possible to dynamically adjust the size of the heap (Tospace) depending on the memory requirements of the program and blocks that become free within an incremental garbage collection cycle can be linked onto the Tospace list for immediate reuse.

In terms of space efficiency, a block allocated copying collector is better than a copying collector that works with contiguously allocated partitions. For example, consider a two-space collector running at 50% residency and
using contiguously allocated memory: if the amount of live data is $n$ bytes, then the total memory use is $4n$ while for a block allocated semi-space collector only $3n$ bytes are used because the allocation area is re-used after each garbage collection.

The three remaining components reside in the second layer of the Storage Manager and are as follows:

- **The Fast Sequential Allocator** — Provides access to contiguous chunks of memory for object allocation, a number of blocks collectively known as the *nursery* which compiled code can fill sequentially with objects. Each nursery block is filled in turn until the nursery is full, at which point the garbage collector is invoked.

- **Out of Band Allocator** — Provides an interface for allocation of chunks of memory in units of words for use when the sequential allocation scheme is inappropriate. An example, of when the sequential allocation scheme is inappropriate is when storage space is required by the GNU multi-precision integer library. Garbage collection *must not* be permitted to occur when memory is being allocated for the libraries use, if it was, and objects were relocated the references held by the library could become inconsistent with the new locations. Such an allocation *always succeeds* and as such should only be used in short bounded sections of (C or Haskell) code where garbage collection is inconvenient.

- **Garbage Collector** — Currently, two types of collector have been/are being implemented into the new rts of GHC, the first is a two space copying collector, whilst the second is a generational collector. The latter will replace the former, for its functionality is emulated by setting the number of generations to one. Two APIs relate to the invocation of the garbage collector:

```c
void GarbageCollect( void ( *get_roots )(void) )
StgClosure MarkRoot( StgClosure *p )
```

The garbage collector is invoked by calling `GarbageCollect` passing a pointer to a function which will be called back to find all possible live
roots. The callback is needed because it is the application, not the Storage Manager, which knows where all the possible roots are.

The \texttt{get\_roots} function is application specific and calls \texttt{MarkRoot} for each possible root, which returns the new location of the object. The garbage collector traverses the reachable graph from each root to find the closure of live data.

### 5.1.2 The Scheduler

The runtime system supports Concurrent Haskell by default [Peyton Jones, Gordon and Finne, 1996], and must therefore manage the concurrent execution of multiple Haskell threads. It is the Scheduler that does this. The Scheduler runs as a single OS thread, and multiplexes this single execution thread among the runnable Haskell threads. Were each thread to be an OS thread then access to a heap object would have to be synchronised using relatively expensive OS support. The threads all share and mutate a common heap and the Scheduler state not only gives the entire state of the Haskell program, but also gives all garbage collection roots. Every thread returns to the Scheduler before the next thread runs, and a thread is only pre-empted when it makes a call to \texttt{ExtendNursery}. This is an API function to the Fast Sequential Allocator that either moves the heap pointer to point to the next free nursery block or fails indicating the heap is full and the Scheduler should invoke the garbage collector.

### 5.1.3 The Interpreter

The interpreter executes byte code (representing a supercombinator of the user program), which is stored in \textit{Byte Code Objects} (BCOs) in the heap and provides the interpreted execution functionality of the mixed interpreted/compiled execution model. A Haskell thread may execute both compiled and interpreted code. If it is executing interpreted code, it examines the heap object that it is about to evaluate. If it is a BCO then the interpreter moves the byte-code program counter to the start of the object and continues execution, otherwise it returns to the Scheduler, requesting compiled execution. The scheduler initiates compiled execution by loading
machine registers with the heap pointer, stack pointer, etc, and branching
to the entry code of the object.

5.2 The GHC Heap

The heap consists of two kinds of object, head normal forms (or values),
and as yet unevaluated suspensions (or thunks).

A value may contain a number of thunks inside it, for example, a list Cons
cell might have an unevaluated head and/or tail. A value which contains
no thunks is called a normal form. Because of the way in which GHC
universally represents both values and thunks, the term closure is used to
refer to them.

When the value of a thunk is required, the thunk is forced and when
forced for the first time, it is physically updated with its value. GHC uses
the self-updating model where the responsibility for performing updates is
placed on code inside the thunk itself. The code to force a closure simply
pushes a continuation on the stack and enters the closure. If the closure
is a thunk, it arranges for an update to be performed when evaluation is
complete, otherwise it just returns its value. No tests need be performed.
The update overwrites the thunk with a value, which therefore must have a
code pointer, because subsequent forces will re-enter the thunk-turned-value
[Peyton Jones, 1993].

The above demonstrates the need for a code pointer to be associated
with the data structures (closures) that represent both data value and thunk
objects on the heap so that entry of an object may be handled correctly,
whether a value or an evaluation and update is being demanded. The represen-
tation of a function value clearly also necessitates the need for an associ-
ated code pointer for it behaves like a suspended computation - when the
function value is applied to its arguments, the computation is performed,
and an update of some sort will occur. The most compact way to represent
a function value is a block of static code (shared by all dynamic instances
of the value), together with its free variables [Peyton Jones, 1993] (the term
closure is usually more narrowly applied to just this representation). A
function value can therefore be represented as a heap object consisting of a
code pointer pointing to the static code followed by (pointers to) the values of the free variables. To perform the function value computation, a reserved register, the environment pointer, is made to point to the closure and the static code executed. The code can access the closure’s free variables by offsets from the environment pointer, and its arguments by some standard argument-passing convention (eg. in registers, on the stack, or in an activation record) [Peyton Jones, 1993].

5.2.1 Universal Representation of Closures

It has already been demonstrated that every type of heap object has code associated with it that must be executed on entry to the object, with a sequence of values possibly following, and it is for this reason that GHC implements a universal representation for closures (see Figure 5.2) and that the STG-machine is a tagless implementation. Because all objects have the same representation, there is no need for a tag to distinguish one kind of object from another.

![Diagram of GHC closure layout](image_url)

Figure 5.2: GHC closure layout.

The first word of a closure is called its info pointer, and this points to the static entry code, that either evaluates the object to weak head normal form or just returns its value. Immediately before the entry code is the object’s
**info table** containing object specific information:

- Closure type — a constant indicating the kind of closure (e.g. function, thunk, constructor, etc.

- Layout information used by the garbage collector indicating number of fields following or objects referenced by the closure — either a tuple pair (pointers, non-pointers) or a bitmap.

- A pointer to the "Static Reference Table" (SRT) for the object (see section 7.4.1).

- Closure flags — boolean flags for quick access to properties of the closure (e.g. is static, is pointed, etc.)

- Various optional profiling, parallel and debugging fields.

It is worth noting, that the info pointer points to the entry code and not to the info table because it requires one less indirection to reach the entry code.

Closures can be located in three different types of logical memory location. static closures reside in statically allocated memory locations and their addresses are globally accessible; dynamic closures reside in heap locations; and stack closures reside at locations within a thread’s stack, these types of closures, unlike the rest are never pointees, that is, they are never pointed to. The reason that the term ‘logical’ was used to describe possible closure locations is because, although stack closures reside within a thread’s stack, the stack is itself a heap object.

### 5.2.2 GHC Heap Objects

When considering their implementation, GHC heap objects fall into four main categories:

- **Pointed objects** represent values of a pointed type, that is, a type that includes bottom (or nil) and they are the only objects that can be lazily evaluated. All pointed objects may be entered by placing the address of the object in the Node register and jumping to the object’s
entry code. It turns out, that this method of object entry is not the only one, (the different entry methods will be explained in a subsequent section of the report), and so this particular one is distinguished by the phrase 'entry ViaNode'. Pointed objects include:

- **Black holes** — When a thunk is entered, its info pointer is overwritten with a black-hole info pointer. If the thunk is re-entered before it is updated, then its value must be dependent upon itself, thus the program has entered an infinite loop, and the error can be trapped and a suitable message can be displayed. Another benefit of black holing is the plugging of space leaks. When a closure referencing an object that is never to be used again is black-holed during its evaluation, the black-hole identifies the closure as containing no pointers so at the next garbage collector invocation, the referenced object may be collected.

- **Constant Applicative Forms (CAFS)** — a top-level expression with no arguments (i.e. an applicative form whose expression is built entirely from function applications, lambda abstraction thereby being absent, with no variables) [Field and Harrison, 1988].

- **Data constructors** — closures representing values that are constructed with algebraic data type constructors.

- **Functions** — closures representing λ-abstractions.

- **Indirections** — When a thunk has been evaluated, it is overwritten with its value. The thunk info pointer is overwritten by an indirection info pointer whose code simply loads the indirection pointer into Node register and enters the new closure.

- **Partial applications (PAPs)** — The application of a defined function to fewer arguments than appear on the left hand side of its definition. Not only do PAPs appear directly in programs, but they also occur as a result of update operations on function valued thunks.

- **Thanks**
• **Unpointed objects** represent values of unpointed type, a type that does not include bottom and can never be lazily evaluated, i.e. they are always evaluated when on construction, the result of this is that they are never updated or entered. A pointer to an unpointed heap object never references a thunk or indirection but points directly to the object itself. Unpointed objects include:

  - *Immutable arrays* — Arrays whose elements cannot be altered, but can only be read from.

  - *Mutable arrays and variables* — Mutable arrays are arrays whose elements (like variables) may be altered or ‘mutated’.

  - *MVars* — Synchronising variables used by Concurrent Haskell. Essentially they are rendezvous points, mostly for concurrent threads.

  - *Foreign objects* — GHC uses a foreign object to reference an external (heap allocated) object that has been passed from outside the ‘Haskell world’, for example a C-allocated pointer is passed and ‘boxed’ up into a GHC heap object. When GHC has finished with the foreign object, the garbage collector invokes a user-supplied *finaliser* function to free the object and the heap allocated memory.

  - Weak pointers — Such pointers may reference an object but *will not prevent* the object from being collected. A common application is the maintenance of a collection of auxiliary information about objects, where the information alone is useless and should not keep the described objects alive [Wilson, 1992].

• **Activation frames** — Objects, representing ‘continuations’, existing only in contiguous chunks on the stack of a thread, are never pointees nor passed are the passed as arguments.

• **Administrative objects** — Objects allocated by the runtime system itself (and never seen by the programmer). They include:

  - Thread state objects (TSOs). Each Haskell thread has an associated TSO which containing its current state information as well
as its stack into which it is ‘packed’ when it is suspended. When the thread is woken up, it is unpacked from its TSO object into machine registers and other memory locations to provide faster access.

- **Byte Code Objects (BCOs)**

- **The stable pointer table**, which maps the stable name of an object to its heap address. If the need arises for a Haskell object pointer to be passed outside of the ‘Haskell world’, for example to a C routine, the pointer needs to be made ‘stable’ so that the garbage collector does not forget that it exists. Although the address of the object may change when the garbage collector runs, its name remains the same.

For further elaboration on the above, see [Peyton Jones, Marlow, Reid, 1999].

### 5.3 The GHC Stack

The new GHC runtime system now supports a single stack, containing both pointers and non-pointers, which grows downwards, towards decreasing addresses.

Like most compilers based on lazy graph reduction, GHC uses the *push-enter* model for function application as opposed to the *eval-apply* model. In the eval-apply model, the function is evaluated, the arguments are evaluated and the function value is applied to the argument. However, function application of the push-enter model is performed by pushing the arguments onto an *evaluation stack* and then *tail-calling* (or entering) the closure of the function.

By attaching layout information to the return address, the activation records may be treated as stack-allocated, but otherwise normal, heap objects. The return address (at the lowest address of the activation record) is like an info pointer, and it is furnished with an info table that describes the layout of the activation record.
Chapter 6

The While and Field
Garbage Collector

6.1 Algorithm Description

The While and Field collector provides a modified Baker implementation for the Spineless Tagless G-machine (creating a Non-Stop Spineless Tagless G-machine) that attempts to reduce the cost of the Baker read-barrier by associating with each object code that evacuates and scavenges it during a garbage collection. The code is generated for each object-type by the compiler: as the code knows the detailed structure of the object (e.g. which objects are pointers and which are immediate values), it executes without interpretive loops and so is particularly efficient. Moreover, by manipulating these code pointers during garbage collection, each object can implement its own individual read-barrier by tracing itself and its arguments if there is a danger of obsolete pointers being propagated. The overhead of such a read-barrier is solely in manipulating these code pointers and is significantly lower than in previous software implementations [While and Field, 1996].

Figure 6.1 shows the necessary closure, its info table and the associated code pointers.

The algorithm begins with evacuation of the root set to Tospace. Each object-type has its own specific evacuation code .EV4 so the evacuation of an object is performed by loading the address of the evacuation code (the
Figure 6.1: Closure layout required by the While and Field collector

- If the mutator enters the closure at the newly side-effect ed info pointer pointing to .MUTSC then this self-scavenging code is executed, the info pointer is reset and the mutator drops through to the standard entry code at .ENTER. This self-scavenging code calls the evacuation code of the top level children it references and places them on the scavenging queue. Because all objects get placed on the scavenging queue, the garbage collector will eventually attempt to scavenge this object. This is undesirable because the object has already been scavenged. The garbage collector always executes scavenging code at the code pointed to by the offset of the info pointer + 1. The collector will therefore execute the code at .SKSC22 (offset .ENTER + 1) which is the skip-scavenge code that simply checks to see if k scavenges have
been performed or garbage collection is complete and either continues scavenging the next scavenge queue object or terminates the collector. In this way scavenging of an object is prevented from occurring more than once.

- If however, the garbage collector scavenges the closure before the mutator enters it, then the collector executes the self-scavenging code at `.SC22` (again offset 1 from the info pointer pointing to `.ENTER`). This code performs the same scavenging operations as the `.MUTSC` code but instead of dropping through to `.ENTER`, it restores the info pointer to `.ENTER` and drops through to the `.SKSC22` code to check whether collection is complete or whether scavenging should continue. Any mutator entry will now occur in the normal fashion with entry to the closure at `.ENTER`.

### 6.2 Pseudo-code Algorithm

The pseudo-code below, using a C style syntax, outlines the While and Field algorithm and *loosely* demonstrates the side-effecting of the info pointer:

```c
/* The mutator always enters and executes the code pointed
 * to by the info pointer
 */
define mutator_entry_pointer info_pointer

/* The garbage collector always enters and executes the code
 * pointed to by the info pointer + 1
 */
define gc_entry_pointer info_pointer + 1

/* When the mutator entry code pointer is altered to point to
 * the new entry code, so the garbage collector entry
 * point must be altered to enter at the mutator enter pointer + 1
 * This is an abstract operation!
 */
```
side_effect_info_pointer( closure, entry_pointer )
{
    closure.mutaror_entry_pointer = entry_pointer;
    closure.gc_entry_pointer = entry_pointer + 1;
}

mutator_create( object )
{
    if ( not enough space )
    {
        if ( Garbage Collector is running )
        {
            abort( out of memory error );
        }
        else
        {
            /* Mutator computation is now paused
             * while the collector runs
             */
            start_Garbage_Collector();
        }
    }
}

if ( Garbage Collector is running )
{
    /* Do k pieces of work at each allocation */
    for ( scavenge_count = 0; scavenge_count < k; scavenge_count++ )
    {
        /* Remove and get next closure to scavenge */
        closure_to_scavenge = head( closure_scavenge_queue )
        if( closure_to_scavenge == NULL ) stop_Garbage_Collector();
        else
        {
            /* Abstract test - this test is not performed the info
* pointer + 1 has been adjusted to point either to
* .SC22 or to .SKSC2
*/
if ( closure_to_scavenge.gc_entry_pointer == .SC22 )
{
    scavenge( closure_to_scavenge, .ENTER );
    /* Info pointer points to .ENTER hence
     * gc_entry_pointer is now at .SKSC2
     */
}
else /* gc_entry_pointer == .SKSC2 */
{
    /* Already evacuated so do nothing */
}
}
}
allocate_memory( object );
}

start_Garbage_Collector()
{
    /* from_space becomes to_space and to_space becomes from_space */
    flip();

    /* For all roots, copy to new to_space */
    for ( root_count = 0; root_count < total_number_of_roots;
         root_count++
    )
    {
        root = get_root( root_count );
        evacuate( root, .MUTSC );
        /* Info pointer points to .MUTSC hence
         * gc_entry_pointer is now at .SC22
         */
*/
}

 evacate( object, mutator_entry_code_pointer )
 {
   /* Copy the object to to_space */
   object_copy = copy_into_to_space( object );
   leave_forwarding_pointer( object, object_copy );
   scavenge_queue_add( object_copy );
   side_effect_info_pointer( object, mutator_entry_code_pointer );
   return object_copy;
 }

 scavenge( object, mutator_entry_code_pointer )
 {
   /* Evacuate first level children */
   for ( child_count = 0; child_count < total_number_of_children;
     child_count++
   )
   {
     child = get_child( closure );
     evacuate( child, MUTSC );
   }
   side_effect_info_pointer( object, mutator_entry_code_pointer );
 }

 mutator_enter_closure( closure )
 {
   /* This is obviously an abstract test, the info pointer
    * either points at MUTSC or ENTER
    */
   if ( info_pointer == MUTSC )
   {

6.2 Pseudo-code Algorithm

scavenge( closure, .ENTER );

/* Info pointer points to .ENTER hence
   * gc_entry_pointer is now at .SKSC22
   * Now drop through to standard entry code at .ENTER
   */

}  
/* Info pointer points to .ENTER so follow it
   * and execute the standard entry code
   */
execute_standard_entry_code();

The collection algorithm, as Baker suggested, also incrementally scavenges the stack using a similar scheme to above. The code for each closure is augmented with a subroutine call before each of its return points, in the same way as .MUTSC precedes .ENTER. The subroutines scavenge the stack frame of the closure, decrement the next return address on the stack and branch back to the ‘normal’ return code [While and Field, 1996]. The (pointer-) arguments on the stack are therefore scavenged as the currently entered closure returns to its caller. The garbage collector also scavenges the stack ‘bottom-up’ at each call to mutator_create() to ensure that the whole stack is scavenged before the collection is complete. Like this stack scavenging technique, the collection scheme also makes provision for the scavenging of an update stack, where update frames, pushed onto this stack when a thunk is being updated by its value, which are no longer required are scavenged.

The While and Field collector provides a very elegant and efficient scheme for the elimination of (the overheads associated with) Baker’s read-barrier. Unfortunately, design decisions made in GHC, the implementation of the abstract Spineless Tagless G-Machine makes implementation of such a collector impossible without significant redevelopment of the GHC compiler and its runtime system; not only is such redevelopment infeasible in the time allocated to the project but it is massively undesirable to the current
developers of GHC. It is believed that such an implementation would serve only to provide benchmark results for a While and Field collector, and would not be incorporated into a release version of the compiler.

- The above collector requires the layout of a closure to adhere to that represented by Figure 7. Before GHC’s runtime system was re-written for optimisations and support of mixed interpreted/compiled code, the layout of a GHC closure adhered to Figure 6.1 except that the .MUTSC and .SKSC22 code pointers did not exist (these were obviously not needed by the stop and copy collector). The initial objective was to compliment the info table with these code pointers, generate the additional code and implement the collector. As has been previously mentioned, the move from GHC-3.0x to GHC-4.0x resulted in elimination of all code pointers from the info table (remember that the info pointer points at the entry code and not the info table) because the developers discovered that fragmented collection code which was hard to optimise resulted. The instruction cache of a machine is such that local blocks of sequential code are stored, thus with non-sequentially located and hence ‘fragmented’ code the instruction cache hit rate will drop and the performance of the system will be degraded.

- The way in which the stack was managed in the old runtime system of GHC-3.0x meant that the stack scavenging algorithm of the collector could not be implemented in this version.

- The new runtime system of GHC-4.0x now manages only a single combined stack as opposed to the addition of an update stack. In fact, this would probably make the implementation easier!

- Finally and quite significantly, the While and Field collector does not propose a method for handling GHC unpointed objects, the closures of which never get entered. These objects generate a heavy penalty, see section 7.4.1.

A modified While and Field collector is therefore proposed for implementation into GHC that still uses a self-scavenging object but provides a
theoretically less efficient implementation, which is hoped however, to still reduce the costs that would be imposed by a read-barrier.

6.3 Performance Implications

Preliminary investigation, through simulation of a While and Field collector, and performed using version 0.10 of GHC, suggested that the scheme adds an overhead of around ‘10% to the execution time of a program’ [While and Field, 1996]. The simulation could not however be described as a fully incremental because it scavenged both the update and return stacks at the flip. A single overhead is placed on the ‘normal’ execution in that, at each allocation, the mutator must test whether or not a garbage collection is in progress in order to determine whether the collector need be invoked. The scheme places three time overheads on garbage collection:

- The info pointer must be adjusted to point to either the evacuate or scavenge code fragment when either of the respective operations are performed.

- A function call is necessary when scavenging a stack frame and the return address of each callee closure that is active during garbage collection must be decremented.

- A count, scavenge_count, must be maintained for the number of closures, stack pointers and update frames that have been scavenged so that when this count hits the scavenge limit, k, the mutator can be resumed.
Chapter 7

An Incremental Self-Scavenging Garbage Collector for *GHC*

7.1 The Spineless Not-So-Tagless G-Machine

As mentioned in section 5.2.1 the info table associated with a GHC closure contains a ‘closure type’ tag. Although the evaluation model of the compiler does not require objects to be tagged in order to distinguish their object-type during evaluation and update, the current garbage collector implementations require this tag to determine the object-type at collection. This is because interpreted evacuation and scavenge code is used where a single routine interprets the object tag and the section of the routine pertaining to this tag is executed (case statements are used). The While and Field collector proposed a non-interpreted scheme where each object was associated with its own evacuation and scavenge code, hence it should be clear that a tag is not required.

The elimination of the code pointers from the info table means that an incremental garbage collector implemented into GHC must also use this interpreted approach. This is theoretically less efficient than object associated code because instead of performing immediate execution of the evacuation and scavenge code, a test is needed to determine the object-type and hence
which segment of code should be executed. In addition to this overhead, there is also cost of the scavenging function — storage of registers, pushing of activation records etc. On the other hand it is not currently known how these costs would compare to the performance degradation that results from the use of the suggested code fragments.

7.2 Algorithm Description

Like both a stop and copy collector and a Baker collector, the collector is initiated when the allocation of a new object fails and ToSpace and FromSpace are flipped (remember, ToSpace and FromSpace are implemented in GHC using a chain of nursery blocks).

In order to simplify explanation of the algorithm, it is convenient to think of the mutator and garbage collector as separate processes and to distinguish between two types of scavenging, collector-scavenging and mutator-scavenging. A collector-scavenge is performed by the garbage collector and is the process of scavenging k objects on the scavenging queue. Mutator-scavenging will be explained shortly.

Having performed the flip, the whole root set is evacuated from Fromspace into ToSpace, k objects are collector-scavenged and the new object is allocated. Evacuation of an object entails copying the object to ToSpace, placing a reference to the copy on the scavenging queue and overwriting the Fromspace object with a forwarding pointer referencing the ToSpace object. It is at this point, that the scheme deviates from Baker's algorithm.

7.2.1 Eliminating the Read-Barrier — The Self-Scavenging Object

At evacuation, after the object has been copied and placed on the scavenging queue, the collector implements the equivalent of the Baker read-barrier, to prevent the mutator from following the references of newly evacuated, pre-scavenged, objects to objects in Fromspace. It does this by overwriting the copied ToSpace object's entry code info pointer with an info pointer pointing to the generic, tagged, scavenging code, .SCAV, that handles all object-
7.2 Algorithm Description

Closure layout seen by compiler and rest of runtime system except for the garbage collector

![Diagram of closure layout]

Figure 7.1: Closure layout required by the self-scavenging collector

types. Because the scavenger code uses generic code and hence needs to test the tag of the object and also know its layout, and because, once scavenging is complete the original info pointer must be restored so that the standard entry code may be executed as normal, a copy of the original info pointer must be maintained. This is done by prepending the original info pointer to the closure (see Figure 7.1). When the space is allocated for the ToSpace copy of the object being evacuated, an extra contiguous word (the size of an info pointer) is required. The info pointer of the closure is then copied into the first word of this returned chunk of memory and the pointer to it advanced by one word. The pointer to the chunk of memory now points to where the closure, and hence the info pointer used by the evaluators, should begin. The closure is copied to the memory chunk beginning at this updated object pointer yielding a self-scavenging object. In this way, it is only the garbage collector, and not the remaining parts of the runtime system or compiler, that is aware of this extra word containing the info pointer copy. The rest of the system can therefore function as normal and the layout of a closure need not be altered.

Collector-scavenging of an object is performed as explained above, by
directly calling the tagged scavenging code and evacuating any (child) objects that it references. Having done this, the original info pointer that is hidden away is now restored, by overwriting the self-scavenging info pointer with the copy that was prepended to the object on evacuation. The garbage collector now pauses and the mutator resumes so that progress can be made. The next time that a new object is instantiated, the mutator is again paused, and another \( k \) objects collector-scavenged before the memory for the new object is allocated from Tspace and the mutator resumed.

Like Baker’s algorithm and the While and Field algorithm, we now examine the use and workings of this read-barrier equivalent mechanism. It should be apparent from above, that if it is the garbage collector that scavenges the object (by performing a collector-scavenge) then the closure is never entered and the .SCAV code is not called. If, however, the mutator enters a pre-scavenged Tspace object, which is therefore still on the scavenger queue, then the mutator follows the closure’s info pointer to what it thinks is standard entry code. This code is not in fact .ENTER, but .SCAV, the self-scavenging code. The .SCAV code causes the closure to be scavenged (and the object to be removed from the scavenger queue) and on completion restores the .ENTER info pointer and follows it resulting in the execution of the standard (original) closure entry code. It is therefore impossible for the mutator to encounter a reference to a Fromspace object, for should it try to by attempting to enter a Tspace object that still makes such a reference, the object scavenges itself. This is the definition of a mutator-scavenge, and it should be noted that on such a scavenger, only the closure that has been entered is scavenged (and its Fromspace references evacuated) and not \( k \) scavenges as performed by a collector-scavenge.

### 7.3 Pseudo-code Algorithm

The pseudo-code below, using a C style syntax, outlines the proposed self-scavenging algorithm and loosely demonstrates the side-effecting of the info pointer:

```c
/* The mutator always enters and executes the code pointed
 * to by the info pointer whether it be standard entry code or
```
7.3 Pseudo-code Algorithm

* self-scavenging entry code
* The garbage collector never enters a closure but executes the
* interpretive scavenger code
*
* The pseudo-code routines described in section 3 describing
* Baker’s algorithm remain the same for our incremental
* self-scavenging collector
*
* The copy_into_to_space routine is, however, modified to
* overwrite the original info pointer with the
* self-scavenging info pointer.
*
/*
* Copy the original info pointer and set the
* self-scavenging info pointer
*/
set_self_scavenging_info_pointer( closure )
{
    /* A temporary pointer for access to the original info_pointer */
    original_info_pointer = closure.info_pointer;

    /* Set the first word to contain a copy of the info table */
    memcpy( closure, closure.info_pointer,
            sizeof( closure.info_pointer )
    );

    /* Update closure to start a word on, we don’t want the system to
    * know about this copy word
    */
    closure += sizeof( closure.info_pointer );

    /* Overwrite the old info pointer with the
    * scavenging code info pointer
    */
memcpy( closure, SCAVENGING_INFO_POINTER,
    sizeof( SCAVENGING_INFO_POINTER )
);
}

/* Restore the copy of the original info pointer */
restore_closure_info_pointer( closure )
{
    /* Restore to the first word of the closure, the original info
     * pointer, hidden in the word directly preceding the closure
     */
    memcpy( closure, closure - sizeof( closure.info_pointer ),
            sizeof( closure.info_pointer )
    );
}

/* Copying an object in to to_space */
copy_into_to_space( object )
{
    /* Allocate memory for object in to_space plus an extra word
     * for the info pointer copy
     */
    object_copy = allocate( to_space, sizeof( object ) +
                          sizeof( object.info_pointer )
    );

    /* Set the info pointer to self-scavenging code
     * and advance object_copy pointer to point to the scavenge_object
     * info table, not the copy of the original
     */
    set_self_scavenging_info_pointer( object_copy );

    return object_copy
}
mutator-enter-closure( closure )
{
    /* This is obviously an abstract test, the info pointer
     * either points at .SCAV or .ENTER
     */
    if ( info-pointer == .SCAV )
    {
        /* Scavenge the closure with a routine that
         * interprets the tag of the object and acts on it
         */
        scavenge( closure );

        /* Restore the copy of the original info pointer to
         * to point back to .ENTER and drop through to this
         * standard entry code
         */
        restore-closure-info-pointer( closure );
    }

    /* Info pointer points to .ENTER so follow it
     * and execute the standard entry code
     */
    execute-standard-entry-code();
}

7.4 GHC Specific Considerations

The algorithm above provides a generalised scheme for implementation into
a Spindless Tagless G-machine, with tagged enterable (pointed) objects and
generic evacuation and scavenging code which interprets the object tags and
acts on them. Incorporation into GHC however, involves further considera-

7.4.1 Pointed Objects — Not the Only Type of GHC Heap Object

Unfortunately, pointed objects are not the only type of object that exist within GHC’s heap, and even then, there are some subtleties that must be taken into account when handling some of them. The following section discusses how the garbage collector handles collection of each type of heap object.

- Pointed objects — For the majority of objects that fall into this type category, garbage collection is performed as per the algorithm above, with objects becoming self-scavenging ‘enabled’ objects at time of evacuation. The following points mention some considerations that are worth noting for collection of specific pointed types-objects:

  - Constant Applicative Forms (CAF$S$) — Represented by closures in static memory that are updated with indirections to objects in the heap once the expression is evaluated. CAFs can’t be reached by following the live roots of the running program: Pointers to static objects are not stored in dynamic memory because at compile time it is known where the object resides and hence there is no need to store pointers to it in the heap. Unfortunately, it is not possible to detect if the CAF expression will be referenced again just by looking at the contents of the heap and stack; one has to look at the code. A static reference table (SRT) is generated for each code segment in the program, and it contains pointers to all of the top level functions and CAFs that the code directly references. For those closures which have an SRT associated with them (some do not, constructors for example — they are fully evaluated) whether the .SCAV code performs the scavenging or the garbage collector does it, the pointers in the SRT are recursively followed at object scavenging time. Any live CAFs that result from this operation are linked, via their link field onto a Tospace ‘live static object list’. The link field is placed at the end of the closure’s payload and points to the next object on the chain.
At the completion of a garbage collection, with all objects scavenged, hence all SRTs recursively followed and live CAFs placed on the list, the CAFs left on the old (CAF) static object list are reverted - an entered CAF (that has been updated) is restored to its original unentered form. Additional support is need to be added to support Hugs which allocates memory dynamically for CAFs, these may obviously, be relocated in a semi-space during garbage collection, whereas static CAFs cannot.

- Indirections — Indirections are not evacuated during garbage collection, they are collected. At collection time, the indirection is followed and the pointer to which it points is evacuated.

- Partial Applications (PAPs) — Are treated like function applications and hence their payload may be thought of as a ‘miniature’ stack. When the object is scavenged the function pointer must be evacuated and the payload can be scavenged in the same way as for the stack.

• **Unpointed objects** pose an unfortunate problem for our scheme because they are not entered and hence cannot be self-scavenging objects. The result is that the garbage collector is not ‘informed’ when they are accessed, for example, in the case of an array when the structure is indexed into, incremental scavenging of such objects could then result in the mutator violating the invariant and obtaining a handle to Fromspace references. For the implementation that supports this report unpointed objects are scavenged in their entirety on evacuation, thus reducing the ‘incrementality’ of the scheme. There are possible schemes that *may* allow these objects to be incrementally scavenged and these are presented in section 10.2.

• **Stack and update frames** (Activation frames) — For the implementation that supports this report, the stack is initially to be treated as a source of roots at the initiation of a garbage collection and all activation records and update frames scavenged in their entirety. There are possible schemes that *may* allow these objects to be incrementally scavenged and these are presented in section 10.2.
• Administrative objects

  - Thread State Objects (TSOs) — If it is a large TSO then it is not evacuated, it is treated as a *large object* (the handling of which is described below) and scavenged as for a small TSO. If not then it is a small TSO, and is evacuated along with the update frame list (thread stack) that it contains and treated as a non-entrant object (for it is unpointed) that must be scavenged at evacuation time.

  - Byte Code Objects (BCOs) — Can be handled as a self-scavenging heap objects, for it is of pointed type. Such an object will become garbage when, for example, a module is re-loaded by the interpreter.

  - The stable pointer table — is treated as a source of roots at initialisation of garbage collection. Since the mutator can access an object pointed to by a stable pointer without entering it, they must be treated as unpointed objects.

The final consideration of an incremental collection scheme for GHC is that of *large objects*. A large object is simply an object whose size is greater than a GHC block (currently 4k, see section 5.1.1) of memory and hence occupies a series of contiguous blocks, i.e. a chunk, that is the right size of the object. The Out of Band Allocator (see section 5.1.1) is used to allocate this contiguous chunk of memory. The White and Field collector made no proposal as to how such objects should be handled, and Baker suggests that they be evacuated lazily. This incurs several overheads (see section 4). GHC however, manages large objects in a different way to those implementations which require large objects to be evacuated. The use of a block allocation scheme can be used to an advantage for a ‘large object list’ can be maintained which contains chained blocks that contain the large objects. When a large object is allocated, the chunk that contains the object is chained on to the list. When a garbage collection occurs, the large object list is traversed (incrementally) and live objects chained on to a new list, i.e they are not copied. Any objects left on the old list may be reclaimed by
the collector. Because the objects are not moved the copying overhead of
the Baker collector, is not present.

The implementation is not expected to cause side effects to such oper-
ations as stack squeezing (that ‘squeezes out’, i.e. removes, empty update
frames from the stack) or lazy black-holing (instead of black-holing every
thunk on entry, wait until the garbage collector is called, and then black-hole
all (and only) the thunks whose evaluation is in progress at that moment
[Marlow and Peyton Jones, 1998]).

7.5 Performance Implications

The self-scavenging scheme incurs the standard overhead associated with
incremental schemes of having to perform a check, to see if the garbage
collector is running or whether the collector should be invoked, at each
new object allocation (call to mutator_create()). However, the test is
eliminated for out of band allocation in the implementation through the use
of a function pointer which is described in section 8.

In comparison to the While and Field collector, the performance of the
proposed incremental self-scavenging scheme is expected to be worse. There
is both a time and a space penalty.

- The While and Field collector executes its evacuation and scavenge
  routines on objects and the stack immediately without having to per-
  form a test on the tag of the object.

- The self-scavenging scheme incurs a space penalty of one word re-
  quired to store the original info pointer of the self-scavenging closure,
  the While and Field collector does not. The time overhead that oc-
  curs when the info pointer of the closure is adjusted at evacuation
  and overwritten, and at scavenging when it is restored, remains. The
  While and Field collector must also adjust its info-pointer at these
times, although the penalty is not so severe because it just offsets it
  as opposed to copying it.

- Like the While and Field collector, closures and stack frames must be
  counted as they are scavenged (in order to test \( k \)).
• Because tagged generic scavenging code is used, there is an overhead of a function call to it whenever scavenging is performed, for both the self-scavenging and Baker schemes; for the While and Field collector, there is not, because the code fragments are jumped to directly. On the other hand, these code fragments impose a time overhead resulting from instruction cache misses (see section 6.1).

• As will be explained in implementation (section 8), a scavenge queue is not used. As a result of this and the processes of mutator-scavenging and collector-scavenging, there is the risk of the the garbage collector scavenging a previously self-scavenged (mutator-scavenged) object. This would not be possible if a scavenge queue were used as only objects that are self-scavenging or on the scavenge queue would be scavenged. It would not be possible for a mutator-scavenged object to be found on the scavenge queue by the collector, since the mutator would remove it on scavenging. Therefore a time overhead will occur whilst the scavengers deduce whether or not the object has been scavenged. The While and Field collector uses a technique that prevents scavenging of an object occurring twice, by side-effecting the info pointer to the skip scavenge code.

• It is hard to know whether the penalty of association of unique evacuation and scavenging code with each object-type (resulting in fragmented and hard to optimise garbage collection code), used by the While and Field collector is going to be greater than the combined effect of the costs mentioned above for the self-scavenging object collector. It is suspected that it will not, but only by implementing a true While and Field collector in GHC, can such costs be compared.

However, comparing the costs of the proposed scheme with a Baker implementation, it is believed that this scheme will prove to be more efficient. The cost of entering a closure is constant (the closure has to be entered by the mutator eventually for both schemes). The question is whether the test of a read-barrier at entry to each closure (which is how a read-barrier must be implemented into pointed GHC heap objects) is more expensive than a
7.5 Performance Implications

scavenge on entry scheme. In order to quantify the difference, during the implementation phase of the project, a Baker collector is also implemented.

It is most unfortunate that the unpointed objects and the stack must be scavenged in their entirety on evacuation leading to only a partial implementation. There are actually however, very few unpointed objects existing within the GHC heap and so the main method of preventing the mutator from making Fromspace references is indeed the self-scavenging object. The performance results will therefore indicate the performance that a While and Field collector will tend towards for a runtime system where such unpointed objects do not exist but the stack is scavenged immediately.

Similarly there are generally very few CAFs within the system and as such, the overheads of having to recursively follow the SRT pointers should be quite small. In [Peyton Jones and Marlow, 1998] they notice that the CAF collection scheme can be optimised, ‘for many code sequences do not reference any static objects at all, and hence have an empty SRT. A top-level function with an empty SRT need not be referenced from other SRTs, possibly resulting in more empty SRTs, and so on.’ As a result, most functions have an empty SRT and those with one, contain very few entries.
Chapter 8

Implementation and Testing

8.1 The Self-Scavenging Collector

The implementation of the self-scavenging collector primarily concerns just the runtime system of GHC. To give an indication of the size of the runtime system, there are twenty eight C source files, six Abstract C source files and sixty one C header files. In total, there are approximately 30,000 lines of code. The majority of the code makes extensive use of macros. The use of macros means that efficient code can be written and the overheads of functions (register storage and stack frame pushes), used to achieve the same task, can be avoided. The object code that results, whilst more efficient is also larger — macros get expanded to the code in their definition by *cpp* the C preprocessor and this is equivalent to the inlining of functions. Therefore, there is a trade-off of efficiency against code size that must be made. The GHC developers have chosen efficiency, the unfortunate result of which is that the code is difficult to read and substantially obfuscated.

The most effective way to implement an incremental collector, especially if the developer is new to the code, is to ‘build’ incrementality into the existing stop and copy collector. This should be done in such a way that each modification can be tested individually as opposed to applying a series of modifications that are all required before a test can be carried out. It should also be ensured that the collector still operates correctly as a stop and copy collector and no maintenance side effects have been introduced.
However, for the purpose of description, it is more intuitive to explain the modifications by mapping them to the flow of execution of both mutator and collector during a collection and not by the order in which they are applied in the development cycle. Figure 8.1 outlines the basic execution flow of the collector.

![Diagram]

Figure 8.1: Basic flow of execution for the self-scavenging collector

### 8.1.1 Occurrence of a Heap Overflow

The garbage collector is invoked by the Scheduler when a running Haskell thread returns to it as a result of the heap being exhausted. To invoke the collector the `GarbageCollect` API is invoked. The code that handles the heap overflow in the scheduler is modified to perform a test to see if the collector is already enabled. If it is, then the heap overflow that has occurred has done so during an incremental collection cycle. The only way of handling this sequence of events is to force the current collection to run to completion, performing non-incremental garbage collection, and then immediately initiating an incremental collection cycle. The danger of doing
this is that unacceptable pauses in the mutator may occur if this happens frequently, and the performance will be massively reduced. To prevent this however, the heap size is calculated using a dynamic sizing policy based on the amount of live data from the previous collection cycle. As a result of this, the collection statistics show that such a ‘force to completion’ has been avoided in all the test runs performed.

The scavenging that results after a heap overflow has occurred is performed by the garbage collector as opposed to the mutator. A new API is introduced to invoke collector-scavenging:

\[
\text{int GC\_incremental\_Scavenge(rtsBool run\_to\_completion)}
\]

The API takes a single argument flag, \texttt{run\_to\_completion}, that indicates whether the current garbage collection should be forced to run non-incrementally to completion. On a heap overflow, this flag is set to ‘true’. The \texttt{GC\_incremental\_Scavenge} API returns an integer value indicating the current ‘step’ that new cells should be evacuated to. Each block descriptor contains a step field used by the garbage collector to identify whether a block of memory is associated with Fromspace or Tospace. During an (incremental) garbage collection, live data is evacuated to Tospace and new cells are allocated to the new heap; any blocks of memory in these areas have a step value of \texttt{TO\_SPACE}, a constant defined as ‘1’, whilst Fromspace (the Tospace and heap of the previous collection) have a step value of \texttt{FROM\_SPACE}, a constant defined as ‘0’. \texttt{GC\_incremental\_Scavenge} returns with a value of \texttt{TO\_SPACE} to indicate that an incremental scavenging cycle is complete (k objects scavenged), but that the garbage collection is itself not complete and a value of \texttt{FROM\_SPACE} to indicate that it is.

Consequently, the garbage collector will operate as a stop and copy collector if \texttt{run\_to\_completion} is set ‘true’ (testing that the garbage collector still operates correctly as a stop and copy collector is performed by always setting this flag to ‘true’ whenever the API is invoked) and \texttt{GC\_incremental\_Scavenge} will always return with a value of \texttt{FROM\_SPACE} if forced to run to completion.
8.1.2 Initiation of the Garbage Collector

When the GarbageCollect API is invoked, the following actions are performed:

- A flag, gc_in_progress is set indicating that garbage collection is in progress.

- The ‘flip’ is performed. In GHC, because allocation of storage space is performed by the Block Allocator a ‘flip’ operation maps to the allocation of a new set of nursery blocks, the number of which is determined by the amount of live data allocated in the previous collection cycle. The nursery is assigned to be referenced by nursery whilst the old nursery by old_nursery. A single block is then requested from the Block Allocator that is used as Tospace. This block is not allocated from the nursery, but is in addition to it. Further requests are made by the collector to the Block Allocator for more Tospace blocks as and when the collector exhausts the free space of the current Tospace block. The step fields of the block descriptors for every block of memory that have just been allocated are set to TO_SPACE.

- k or scavengelimit, the limit on the number of objects that are permitted to be scavenged by the collector before the mutator can be resumed is (ensuring only the necessary amount of scavenging is performed) is determined using Equation 3.3.

- All the roots that of the application are now evacuated to Tospace.

- It is possible that the garbage collector has been invoked directly from Haskell as opposed to by the Scheduler. If this is the case, the current thread must be paused and also evacuated.

- The ‘weak pointer list’ that contains references to all the (live) weak pointers that the application has must be traversed and each weak pointer evacuated. Essentially, they are treated as roots of the system. This must be performed so that dead weak pointers can be detected during collection and their finalisers run.
8.1 The Self-Scavenging Collector

- All the CAFs referenced by the CAF list must be treated as roots and evacuated, the details of CAF garbage collection will be explained further in the chapter.

- A call is made to GC\texttt{incrementalScavenge} API to collector-scavenge \texttt{scavenge\_limit} objects.

When GC\texttt{incrementalScavenge} returns, the mutator is resumed and the allocation of the object that previously resulted in invocation of the collector is now performed.

Before detailing the subsequent flow of execution, the operation of the \texttt{evacuate} routine must be explained:

8.1.3 Evacuating an Object

The \texttt{evacuate} routine gets called \textit{eventually} for every live object in the system. There are only two major modifications that need be made to this routine:

- The code that handles the evacuation of each unpointed object must be complemented with a call to the appropriate scavenging routine. In this way, by scavenging the unpointed object in its entirety upon evacuation, the mutator is unable to violate the invariant and obtain a reference to a Fromspace object.

- The code that handles \textit{thunk selectors} must be altered. A thunk selector is a dynamically allocated thunk whose entry code performs a selection operation from a data constructor drawn from a single-constructor type. The garbage collector can perform an optimisation here. It looks at the tag in the info table of the thunk selectee. If the selectee is evaluated, then it performs the selection, and then continues as if the selector thunk is an indirection to the selected field. If it is not evaluated, it treats the selector thunk like any other thunk of that shape. The problem that this presents to the incremental collector is the case where the selectee is a self-scavenging object: looking at the tag in the info table of a self-scavenging object will not correctly indicate whether the selectee is evaluated or not. The code must therefore
be altered to examine the tag in the info table pointed to by the info pointer copy that is prepended to the closure.

When a pointed object is evacuated and thus copied to Tamespace, it must be ‘redefined’ as a self-scavenging object. The following describes how a self-scavenging object is defined:

**Defining the Self-scavenging Object**

In order to create the self-scavenging object it must have the required data structures defined; both the compiler and the runtime system will make use of these and therefore the structures are defined in the header files that are accessible to both systems (found in the includes directory). The two structures below show the generic format that all GHC closures follow:

```c
typedef struct {
    const struct _StgInfoTable* info;
    StgProfHeader     prof;
    StgGranHeader     par;
    StgTickyHeader    ticky;
} StgHeader;

typedef struct StgClosure_ {
    StgHeader   header;
    struct StgClosure_ *payload[0];
} StgClosure;
```

Focusing on the second structure, it contains two elements: the standard (fixed size) closure header and a pointer to the start of the object’s payload. The standard GHC closure header defined by the structure StgHeader contains a pointer to the info table of the object (see section 5.2.1 for a description of the elements in an info table), and three other headers used by different builds of GHC. These are variants of the ‘vanilla’ build of GHC that is used in the supporting implementation of this project. They are: the profiled variant, the parallel GHC compiler and the ticky profiler. For the ‘vanilla’ build, none of these three structures are used; nor are they of
any relevance and as such they will not be referred to again. The payload
pointer points immediately after this info pointer structure to the fields that
are associated with the closure.

The self-scavenging object is defined as a minimal closure following
the standard format defined above and so contains the StgHeader element
header, but unlike most other objects, nothing else. The declaration is
defined below:

typedef struct {
    StgHeader  header;
} StgSelfScavenging;

Because the original info pointer of the object that has just been evac-
uated will be prepended to the closure, i.e. in front of the self-scavenging
info pointer, all the original layout information is preserved for use by the
(mutator- or collector-) scavenger when scavenging occurs. There is there-
fore no need for the StgSelfScavenging info pointer to carry any other
information, or have its layout information set. Essentially, all that is of
interest for the purposes of scavenging is whether or not the object is self-
scavenging, indicating that it is yet to be scavenged, or a scavenged object
with its restored info pointer, this can be determined by the type field in
the info table.

Each type of closure has a base identifier that acts as a naming conven-
tion for all of the structures associated with it. The base identifier for the
the self-scavenging object is defined in the implementation that supports
this project as SELF_SCAVENGING. The type (or tag) of an object is defined
as an integer constant using just its base identifier and its value should be
that of the last closure currently declared in the ClosureTypes.h header
file, incremented by one. Having defined this constant, it must be added
to the StgClosureType enumeration structure. The only other field of the
info table that must be initialised for a self-scavenging object is that which
contains the closure flags that are used for quick access to properties of the
closure. The closure is minimal. It therefore has no properties (it has no
SRT, nor is it in head normal form or static or a thunk or mutable etc.),
so \texttt{FLAGS\_SELF\_SCAVENGING} is set to zero. All that remains is for the forward declarations for the info table and entry code to be defined in the shared header files and the actual structures defined in Abstract C in the file \texttt{StgMiscClosures.hc} that belongs to the runtime system. The naming convention for the object is adhered to, thus yielding:

\begin{verbatim}
SELF\_SCAVENGING\_info    /* the info pointer */
SELF\_SCAVENGING\_entry   /* the entry code */
\end{verbatim}

The \texttt{INFO\_TABLE} macro is used to define an info table. In the example below, it defines the (\texttt{SELF\_SCAVENGING\_info}), associating it with the \texttt{SELF\_SCAVENGING\_entry} code, setting the payload information (number of pointers and non-pointers) to zero, the type to \texttt{SELF\_SCAVENGING}, indicating that the information it carries is constant and will not change. The remaining fields are not relevant.

\begin{verbatim}
INFO\_TABLE(SELF\_SCAVENGING\_info,SELF\_SCAVENGING\_entry, 
0,0,SELF\_SCAVENGING,\emph{const},\texttt{EF},0,0);
\end{verbatim}

The \texttt{SELF\_SCAVENGING\_entry} (entry) code for the self-scavenging object is defined using the \texttt{STGFUN} macro that declares it as an STG-machine function:

\begin{verbatim}
STGFUN(SELF\_SCAVENGING\_entry)
{
    \texttt{FB_}
    \texttt{CALLER\_SAVE\_R1}
    \texttt{STGCALL1(MutatorScavenge,R1.p)};
    \texttt{CALLER\_RESTORE\_R1}
    \texttt{JMP\__(GET\_ENTRY(R1.c1))};
    \texttt{FE_}
}
\end{verbatim}

The function works as follows. When the mutator enters the closure it executes the \texttt{SELF\_SCAVENGING\_entry} code, saving register R1 to prevent it from corruption, calls the \texttt{MutatorScavenge} C routine with R1 passed as an
argument, then restores R1 before jumping to the entry code of the closure that it points to. The scavenging operations of the MutatorScavenge routine is explained in greater detail later on in this chapter. At the initial stage of the implementation, all it does is restore the info pointer copy, overwriting the SELF_SCAVENGING_info info pointer. It is the entry code pointed to by the restored info pointer that is then jumped to.

The implementation of a self-scavenging object is complete, all that needs to be done is the modification of the routine that allocates space for the object that is being evacuated to Tospace. The first modification is to amend the allocation request for an extra word, the size of an info pointer, this is used to store the closure’s original info pointer. The info pointer of the object is copied into the first word of the returned storage space, the second word is set to the address of SELF_SCAVENGING_info, the self-scavenging info pointer, then the payload of the closure is copied into the remaining space starting at the third word. The address that is returned by this routine must be that of the first valid word of the closure, that is the second word of the allocation space, i.e. the address of the location housing the self-scavenging info pointer. The Fromspace object that has now been evacuated to Tospace is overwritten with an object indicating that it has already been evacuated and its forwarding address is set to the address returned by the evacuate routine. It is an error for the mutator to ever enter an ‘evacuated’ object as this violates the invariant that the mutator can should never see a Fromspace reference. It is the scavenger that uses these forwarding addresses so that additional references from objects that did not initiate the original evacuation can be updated. The result of this is that the rest of the runtime system is unaware that the prepended info pointer copies exist. The scavenge routine must now be updated to handle the self-scavenging objects, otherwise, as it traverses Tospace, it will hit the info pointer copies first and attempt to evacuate what is at the address of the SELF_SCAVENGING_info; this will almost certainly result in an assertion failure, for the evacuate routine performs checks on the closure it is about to evacuate. The modifications to this scavenging routine is explained in a subsequent section that also details the other scavenging routines.
The description of the flow of execution within the execution cycle is now continued. As the mutator executes, it allocates new objects, and it is at these allocations that the collector performs an incremental scavenge of Tospase:

8.1.4 Collector Scavenging

The `GCinCREMENTALSCAVENGE` API is not only invoked at the initiation of garbage collection or after a heap overflow during a collection, but also every time the mutator allocates a new object in the heap, or when storage space is requested from the Out of Band Allocator. In the implementation, it is convenient to place the call in the code which checks that allocation of a new object in the heap is possible, this requires that a test be performed in this code to see if garbage collection is in progress. On the other hand, the test is avoided in the allocation routine of the Block Allocator through the use of a function pointer. If `gc_in_progress`, then this pointer points to a function that performs that calls `GCinCREMENTALSCAVENGE` and performs the allocation. If the collector is not enabled, then the function pointer points to the standard Block Allocator routine.

![Figure 8.2: Control flow of GCinCREMENTALSCAVENGE](image)

`traverse_weak_ptr_list`
8.1 The Self-Scavenging Collector

If GCinremerntalScavenge is invoked as the result of an allocation of a new object, then the routine performs an incremental scaveng cycle (scavenging \( K \) objects). Such a cycle involves the scavenging of all possible types of objects within the system. As a result GCincrementalScavenge loops, while the (current) scaveng_count is less than the scaveng_limit or if run_to_completion is 'true', while there still exist live objects in the system that are yet to be scavenged. Within this loop, the four scavenging routines, that between them handle all the different types of objects in the system, are invoked (see Figure 8.2):

- To space, using the scaveng routine.
- Statically allocated (fixed address) objects, using the scaveng_static routine.
- The large objects list using the scaveng_large routine. Large objects reside in their own storage area and are never evacuated from it, but are always scavenged.
- The weak pointer list, using the traverse_weak_ptr_list routine.

The scavenging operations of these routines are explained in subsequent sections.

8.1.5 Mutator Scavenging

Whilst executing, the mutator may enter a self-scavenging object. This object must be scavenged before the mutator is allowed to handle any object references in the payload. The entry code that is executed when an object of this type is entered has already been detailed. The result of executing the entry code is a call to the MutatorScavenge function that scavenges the object.

The MutatorScavenge routine is not optimal in the implementation, for it imposes several function call overheads that could be eliminated. Two of the function calls made are used to measure performance metrics of the collector, more specifically how long the mutator-scavenge takes. The other is a call to the scaveng function, the generic scavenging routine
used whether the scavenging operation is a mutator-scavenge or a collector-scavenge. The address of the closure that has just been entered, (the self-scavenging object) was passed in to the function from register R1. All that the MutatorScavenge function does is pass its argument to the scavenging routine indicating that only a single object should be scavenged (and not up to the scavenging limit). A call must also be made to the scavenging static routine that scavenges static objects. The reason for this will be explained in the section pertaining to the scavenging of static objects. Having done this, the self-scavenging entry code is returned to; This code jumps to the entry code of the restored info pointer.

8.1.6 Termination of Garbage Collection

It should be apparent that mutator execution and mutator-scavenging of objects is interleaved with the collector-scavenging of objects that has already been described. Eventually, in a call to GClcrementalScavenge, all the live data in the system will have been scavenged and the collection cycle is terminated, allowing for ‘pure’ mutator execution (although it is accepted that there is very little of this when an incremental collector is used). The completion of collection is performed by freeing up the blocks allocated to FROM_SPACE; garbage collecting any CAFs (this will be elaborated on in a later section); tidying up various structures, for example setting block descriptor step values for all the newly allocated blocks to FROM_SPACE and terminating lists with ‘nil’; restoring the Out of Band Allocator allocation function pointer to point to the function that does not invoke GClcrementalScavenge on a new object allocation; calling any pending weak pointer finalisers for dead weak pointers; and finally bumping the garbage collection statistics.

This completes the description of the mutator and collector flow of execution within a garbage collection cycle. The scavenging routines and the modifications that support incremental scavenging are now described.
8.1.7 The Simulated Scavenge Queue

Before describing the changes that must be made to the `scavenge` routine, the handling of the scavenge queue must be explained.

GHC's collector does not have a scavenge queue. GHC uses block allocated chunks of memory. The benefit of this is that Tospace is always separate from the heap and Out of Band allocator storage areas. As a result, there is no need for a scavenge queue, since all the objects in Tospace must eventually be scavenged, and there is no danger of the collector scavenging newly allocated objects that are allocated during resumed mutator execution.

It is because the live objects of the system do not end up in a contiguous Tospace chunk, as happens in Baker's algorithm, that incremental scavenging is somewhat more complicated. This is why `GChomcrementalScavenge` loops executing four separate routines, as described in section 8.1.4 above, where each routine may result in the evacuation of objects whose scavenging must be handled by one of the other routines. Garbage Collection is therefore only complete when a pass of the loop is made in which no scavenges are performed and hence no evacuations occur which would result in further objects to be scavenged.

On initiation of garbage collection, once the root set has been evacuated, a pointer, `scan` is set to point at the beginning (i.e. the first object) of Tospace. As collector-scavenge (`GChomcrementalScavenge`) operations are invoked, objects in Tospace are scavenged. Scavenging of objects is, in this way is performed, by scavenging the Tospace object pointed to by `scan` and then advancing the `scan` pointer by the size of the closure that has just been scavenged, to point at the next closure for scavenging. Because the `scan` pointer makes a single pass of the heap, all live objects currently evacuated to Tospace will have been scavenged when `scan` reaches the end of the last Tospace block and it is not possible for the collector to scavenge an object that it has already scavenged. This corresponds to scavenging an object at the head of the scavenge queue and then removing the object from the scavenge queue so it will not be subsequently scavenged.
8.1.8 Scavenging Tospace

The scavenging of Tospace is performed by the scaveng function and there are three main modifications to this routine that are applied in making the transition from a stop and copy to an incremental collector.

- Support for self-scavenging closures. This is further subdivided:
  - This routine is used for the generic scavenging of all Tospace objects, whether invoked by the mutator or the collector. If the collector invoked it, then up to scavengelimit objects need to be scavenged. Alternatively, if the mutator invoked it, then only this single object, (and no more, unless any of its references are unpointed) should be scavenged. The function declaration is therefore modified to take a flag indicating whether the mutator or the collector invoked it in addition to a pointer pointing to the current position of scan.
  - The routine consists of a loop for the iteration over all the objects in Tospace within which is a switch statement that tests the type field of the info pointer of the current object being scavenged and executes the appropriate piece of code. To this must be added a case that handles the self-scavenging object. This piece of code simply copies back the original info pointer, increments scavengecount and then goes round the loop again so that the appropriate scaveng code is executed depending on the type field of the restored info pointer.

- The loop test that determines when the function is exited must be altered. If a collector-scaveng is in progress, the loop must be exited after scavengecount - scavengelimit objects have been scavenged and after one object has been scavenged if a mutator-scaveng is in progress.

- Unfortunately, it is still possible for the collector to hit objects that have already been scavenged as it traverses Tospace. Consider a self-scavenging object that is entered by the resumed mutator, resulting
in its scavenging. The \texttt{scan} pointer has not yet therefore reached this object, for if it had, a collector-scavenge would have occurred and the self-scavenging object would not be entered. To prevent the object being re-scavenged, a set of macros are defined for each type of object that allow its size to be determined and the collector to advance \texttt{scan} to point at the next object in Tospace (i.e. to skip the already scavenged object). These macros, unlike the set used by the stop and copy collector to calculate the size of a closure and advance \texttt{scan} to the next closure beyond the payload ‘know’ about the prepended info pointer copy. It should be obvious that the collector can deduce that an object has already been scavenged if it does not have a self-scavenging info pointer (as a result all heap closures, including unpointed objects have the extra info pointer word prepended to them). There is however a further complication that results from possible scavenging by both the mutator and collector. Consider again the entry of a self-scavenging object by the mutator, and the \texttt{scan} pointer that has yet to pass this object’s position in Tospace. Once the mutator-scavenge has occurred subsequent mutator execution, \texttt{can} result in the object being overwritten with a smaller sized closure. This can happen because steps are taken to evaluate the object to head normal form. If this happens, then when, on a collector-scavenge, the \texttt{scan} pointer hits this updated object and tries to skip the object to reach the next closure, it will fall short, most likely causing a segmentation fault as it tries to resolve the info pointer for the garbage that it believes is the next object. Fortunately, the original info pointer, that is, the info pointer of the larger closure, is still stored in the info pointer word prepended to the closure. If the scavenger is therefore required to skip an object because it has already been scavenged, it must use this info pointer copy and not the current info pointer of the closure. The aforementioned macros are therefore written with this in mind.
8.1.9 Handling Large Objects

In the stop and copy collector large objects are not allocated from the heap. Because each large object takes up one or more blocks, they are created by making a direct request to the Block Allocator for the number of blocks required. The block descriptor that is returned is then chained onto a singly linked list \( \text{large_alloc_list} \), which contains references to the block descriptors of every live object. When a large object is evacuated, its block descriptor reference on \( \text{large_alloc_list} \) is moved to the \( \text{new_large_objects} \) list where it awaits scavenging. On scavenging, its block descriptor references is moved to the \( \text{scavenged_large_objects} \) list. The benefits of this are that the overhead detailed in see section 4.3 does not exist and large objects are never copied to new blocks. At the end of garbage collection, any large objects on the \( \text{large_alloc_list} \) are no longer live and can be have their blocks returned to the list of free blocks maintained by the block allocator. The \( \text{scavenged_large_objects} \) list then becomes the \( \text{large_alloc_list} \).

With the addition of an incremental collector, it is no longer sufficient to use singly linked lists as the evacuation or scavenging of a large object that is not at the head of the list may well be required. It would be exceptionally inefficient to chase down a singly linked list searching for the object that precedes the object being removed from the list so that the link can be maintained once the object is removed. It is preferable to maintain 'last' and 'next' pointers to the object’s predecessor and successor so that the object can be removed from the list and using these fields their links can be updated.

As with the (scavenge) routine, the \( \text{scavenge_large} \) routine must be modified to handle the scavenging of just a single object, or of \( \text{scavenge_count} - \text{scavenge_limit} \) objects.

8.1.10 Handling Weak Pointers

\( \text{traverse_weak_ptr_list} \) is called possibly many times during garbage collection in the main loop of \( \text{GCincmementalScavenge} \). Its order in this loop is essential and it is critical that the following invariant is maintained:

\( \text{traverse_weak_ptr_list} \) is called when the heap is in an idem-
potent state. That means that there are no pending evacuate/scavenge operations. This invariant helps the weak pointer code decide which weak pointers are dead - if there are no new live weak pointers, then all the currently unreachable ones are dead.

The routine returns a flag indicating whether or not it called evacuate on any live pointers.

The self-scavenging collector allows the list to be traversed incrementally until the scavenger_limit is hit. The mutator will then be resumed. Because weak pointers are only removed from this list if they are live it is undesirable to keep scavenging the list from the top each time this routine is called. The position that the scavenger had reached is therefore maintained across calls to this routine so that traversal may be continued from where it was last left off.

On completion of garbage collection, any elements still on the list are dead and may have their finalisers run. Unfortunately, it is possible for mutator computation to complete before the current garbage collection completes. The result, is that the program may exit without the finalisers of the dead weak pointers being run. This can have a disastrous consequences, for example the output weak pointer, that is associated with stdout needs to have its finaliser run in order to dump the result of any computations performed to the screen or terminal. To ensure that the finalisers of such weak pointers are executed, if a garbage collection is in progress when the last/main thread finishes executing, the garbage collection is forced to run to completion before the program may exit.

8.1.11 Scavenging Static Objects

The only type of static objects that point to the heap are CAFs (static indirections). The remaining static objects point only to other static objects either via references in their payload for static constructors or via the SKT for static functions and thunks. Static objects are evacuated by placing them on the static_object list. When the scavenger_static routine is called the objects are removed from this list and placed on the
scavenged_static_objects list. If the object has an SRT, its SRT references are evacuated, its payload references are then evacuated.

Because static objects cannot be set to self-scavenging objects (where would the info pointer copy go!) and also because they are so few in number, there is little benefit in devising an incremental scavenging schemes for this list. The result of this, is that the scavenges static routine must be invoked just before mutator execution is resumed (i.e. in MutatorScavenge and GcincrementalScavenge).

When handling CAFs, the problem is how to find all the static indirections before the mutator can get a reference to them. The static object list is built up during garbage collection, and it's only at the end of collection that it is known where live static objects are. This is a real problem. In order to resolve the problem, a list of all the live CAFs must therefore be maintained (caf_list). The function newCaf does this. At the start of garbage collection, all the CAFs currently on caf_list are treated as roots and their indirectees evacuated; they must not however be placed on the static_objects list. When a static indirection is evacuated, it will get placed onto the static_objects list and eventually after textttscavenge,static has been invoked, it will be placed on the scavenged_static_objects list. At the end of collection, it is known which CAFs are live — they are on the scavenged_static_objects list and their STATIC_LINK fields are non-nil. Any CAFs on caf_list that are not on the static object list can now be removed and overwritten with a blackhole.

In order to stop a new CAF that results through resumed mutator execution from being blackholed, it must be placed immediately on the scavenged_static_objects list. A problem that results with this scheme, is that it is possible for a static object that has become a CAF to already exist on either the scavenged_static_objects or static_objects lists. To prevent these lists from being corrupted and to resolve this problem, the UPD_CAF macro must be altered to preserve the STATIC_LINK field when the the static object that is being updated to a CAF has its info pointer overwritten by a CAF info pointer. If this were not done, the lists could be corrupted because the STATIC_LINK field is initially nil when the new info pointer is written.
8.1 The Self-Scavenging Collector

8.1.12 Closure Entry

The foundation to the self-scavenging collector is that references in the payload of a closure can be only be accessed by entering the object and executing the entry code. This is essential to the invariant. Were this to be violated, and the fields to be accessible without execution of the entry code, the invariant could be violated, with the mutator obtaining Fromspace references. Unfortunately, this invariant can in fact be violated in GHC. GHC distinguishes between two different entry conventions: slow entry and fast entry.

GHC is written with platform independence in mind, and as such a major problem that must be contended with is the use of registers in the call and return conventions. It is cheaper to pass arguments and results in registers than on the stack on a machine which makes available a large number of registers. Alternatively, on a machine with a limited number of registers, it is cheaper to pass arguments and results on the stack than use ‘virtual registers’ in memory. GHC uses a hybrid system with the first n arguments or results passed in registers and the remainder on the stack.

The slow entry point is used when the function does not have all of its arguments in registers. An argument satisfaction check is therefore performed that determines whether the remaining arguments are on the stack and checks they are not obscured by any update frames lying above them. If the satisfaction check succeeds, then the function has all its arguments available and the fast entry point is jumped to. If the satisfaction check fails, then a PAP is constructed containing the arguments that are present on the stack. The fast entry point is used when the function has all its arguments. The pointer to the closure associated with the function is passed in arg₁ and the remaining arguments in arg₁...argₘ₊₁. The pointer to the closure is used to access the free variables of the closure.

The self-scavenging entry code replaces the slow entry code of the object that it overwrites. If the function has a closure (i.e. is not top-level) then a jump straight to the fast entry point must not be allowed. If it were, the self-scavenging entry code could then be bypassed and, as the free variables of the closure are accessed, the mutator may get hold of Fromspace references. This
violates the invariant. Unfortunately, GHC does allow this jump straight to the fast entry point if the argument satisfaction check need not be performed. To resolve this problem, the compiler must be patched so that the code that generates a tail call to a closure always uses the slow entry point entry convention.

This scheme is not however fully optimal, for if the closure is a top-level function then a jump to the fast entry point should be allowed.

### 8.2 The Baker Collector

A Baker collector can be implemented into GHC using most of the framework that the self-scavenging collector provides. This does not however, provide the most optimal Baker implementation in terms of space overheads. A more efficient implementation is suggested in section 10.2.

The major differences between the two implementations are described in the following sections:

#### 8.2.1 Evacuating an Object

![Closure layout seen by compiler and rest of runtime system except for the garbage collector](image)

Figure 8.3: Closure layout used by the Baker collector.

When an object is evacuated to Tospace, the extra word that is prepended to the closure is retained. The original info pointer of the object is copied
as before into this word. However, the address stored at location has ‘1’ immediately subtracted from it (see Figure 8.3. The word that is the first ‘valid’ word of the closure, and hence contains the info pointer used by the evaluators, unlike the self-scavenging implementation, is not updated with a self-scavenging info pointer; it is left containing its original info pointer. The reason that this info pointer copy word is still required is to resolve the update problem in which a closure is updated with one of a smaller size. As collector-scavenging of TOSPACE progresses, this mechanism allows the scan pointer to be set to the next valid closure in TOSPACE see section 8.1.8.

8.2.2 The Read Barrier

To implement a read barrier into GHC that is equivalent to trapping the ‘pointer loads’ mentioned in sec 4.1, closure entry must be trapped using the following macro. Notice that the mutator is prevented from obtaining FROMSPACE references by trapping entry to objects that are either FROMSPACE and evacuating them to TOSPACE as well as trapping entry to unscavenged objects in TOSPACE:

```
#define ENTER_NODE_WITH_ARGS(closure) \ 
closure = RET_STGCALL1(StgClosure *, BakerReadBarrier, \ 
                         (StgClosure*)closure); \ 
JMP_(GET_ENTRY((StgClosure*)closure));
```

The address of the closure that is hit by the barrier is passed via the macro to the BakerReadBarrier function defined below.

```
StgClosure *BakerReadBarrier(StgClosure *p) 
{
    if ( !gc_in_progress ) return p;

    stat_startMutScav();

    if (IS_USER_PTR(p)) {
        bdescr *bd;
```
if (Bdescr((P_)p)->step == FROM_SPACE) {
    p = evacuate(p);
}

if (IS_USER_PTR(p)) {
    if (Bdescr((P_)p)->step != NO_NEED_TO_SCAVENGE) {
        scaveng((StgClosure*)p - sizeof(StgClosure),
                SINGLE_OBJECT_SCAVENGE);
    }
}

if ( static_objects != END_OF_STATIC_LIST ) {
    scavenges_static();
}

stat_endMutScav();
return p;
}

This function evacuates the object if it is in FROM_SPACE, scavenges it if necessary and then returns the address of the evacuated object (or the same location if evacuation was not required) so that its entry code can now be jumped to. Because all closures that are being entered are now checked by the read barrier it is necessary to define a third value for block descriptor step fields, NO_NEED_TO_SCAVENGE. This value is assigned to the block descriptors of those blocks to which the mutator allocates new objects and as a result do not need scavenging. This applies particularly to the nursery, which we cannot distinguish easily from Tospace using just the TO_SPACE identifier.

Having defined the read barrier, all occurrences of jumps to the entry code, of closures must be replaced with the ENTER_NODE_WITH_ARGS. These jumps occur in two forms throughout the runtime system and the compiler (outlined below). It is not however, always immediately apparent if this code
is entering a closure or performing some kind of return (direct, vectored, etc):

\begin{verbatim}
JMP_(GET_ENTRY(R1.c1));
JMP_(ENTRY_CODE(*R1.p));
\end{verbatim}

### 8.2.3 Scavenging Tospace

Section 8.2.1 described the subtraction of ‘1’ from the address stored in the info pointer copy word resulting in an odd address stored at this location. When the object is scavenged, ‘1’ is added to this address, restoring it to its original value. This scheme enables the scavenger to identify and thus skip those closures that have already been scavenged.

### 8.3 Testing

The nofib suite [Partain, 1993] has been developed for the explicit testing and benchmarking of the GHC compiler. This benchmark suite contains over eighty test programs which result in rigorous testing of the compiler. The suite is supplied with scripts for the automated compilation, execution and reporting of statistics for these programs as well as transcripts of the expected output against which the results from the execution of each program are compared.

Both the incremental and Baker garbage collectors have successfully executed the nofib suite, verifying at least the correctness of the implementations.

The relative performance of the schemes, and their performance relative to GHC-4.01’s two space collector, is the subject of the next chapter.
Chapter 9

Benchmarking

9.0.1 The Benchmark Programs

For the purposes of benchmarking, three programs were selected from the nofib test suite:

- Circsim
- Compress
- Wave4main

The three collectors to undergo benchmarking were: GHC-4.01’s existing stop and copy collector, the self-scavenging collector and the Baker collector. Two different builds of the compiler running with the stop and copy collector were created. The first, the ‘vanilla’ build that allows access to closures through fast entry points (see section 8.1.12; and the second that forces entry to a closure to always be via the slow entry code, hence fast entries are disabled.

These programs are selected because, of all the programs in the suite, their execution times are larger than most, but are suitably spread in duration; they perform enough garbage collections for meaningful results to be recorded; and their execution times are not so long that it makes the completion of benchmarking unachievable in the time allocated.

The machine used in benchmarking tests is specified as follows:

- Processor — Pentium II running at 350MHz.
• Memory — 128 Megabytes of DRAM.

• Operating system — RedHat Linux 5.2. A minimal installation with no unnecessary daemons running or modules plugged in.

• Operations are performed from the command line in single user mode. No X Windows server has been installed.

• Hard disk — 4 Gigabyte drive with 125 megabytes allocated as swap space.

• Network capability — removed, and machine physically disconnected from the network.

The tests applied are described below, as are the results. Where times are shown, they are given in seconds with millisecond accuracy. Because only millisecond accuracy is viable, calculations are performed using two decimal place accuracy and calculated values are rounded (not truncated) before being presented.

9.0.2 ‘Raw’ Execution Times

The machine on which the tests are executed has already been ‘stripped to it bear essentials’. This is to ensure that the benchmarks are not affected by the reduced performance of a machine which is managing and concurrently executing processes for other users. Interleaved execution with other processes degrades performance through the flushing of caches that result in cache misses and in turn page faults. In addition to this, the time that is devoted entirely to the execution of a user process (for example one of the test programs) is used to measure the ‘raw’ execution time of the program. This is known as user-time. The elapsed time of the system is not used, for this includes not only the user time but also kernel execution time and the time taken to perform context switches.

When performing benchmarks it is necessary to ensure that the user-time only includes the pure computation time of the code being that is under investigation, for the purpose of this project, this must only be the time that the mutator and garbage collector spend in execution). As a
result of this, any code that performs profiling operations on code being benchmarked needs to be removed. The first set of program runs are used to obtain the raw execution time of the program. The profiling code for the garbage collector is therefore physically removed from all the collectors and they are recompiled. Each of the test programs is then compiled and executed with the appropriate build of the compiler and runtime system for the collector being benchmarked. Whilst every attempt has been made to ensure that the benchmarks are both accurate and consistent, a second execution run of a program will almost invariably yield a *slightly* different result. Therefore, when a program is benchmarked, five timed executions are performed and the average user-time is presented.

Having benchmarked the raw execution times of the collectors, the profiling code of the garbage collectors is enabled so that collection statistics can be gathered. These statistics can be used in to verify the operation of the collector and make comparisons of performance.

### 9.0.3 Benchmark Results

The following section presents the results of raw execution and profiled execution of the test programs.

- Table 9.1 presents the timing results from the raw executions for the stop and copy collector builds, whilst
- Table 9.2 presents those obtained for the incremental collector builds.
- Table 9.3 indicates for each program how many garbage collections occur.
- Table 9.4 presents the profiling statistics for the Ciresim test program.

The items in the table represent the following:

- **INIT** — The garbage collection initialisation time.
- **MUT** — Total time mutator spends executing.
- **GC** — Total time that is spent garbage collecting.
- **MUT SCAV** — Total time that is spent mutator-scavenging.
<table>
<thead>
<tr>
<th>Test program</th>
<th>Stop Copy collector</th>
<th>Stop Copy collector — no fast entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ciresim</td>
<td>31.70</td>
<td>31.75</td>
</tr>
<tr>
<td>Compress</td>
<td>08.58</td>
<td>08.55</td>
</tr>
<tr>
<td>Wave4main</td>
<td>11.60</td>
<td>11.66</td>
</tr>
</tbody>
</table>

Table 9.1: User-times for the (non-profiled) stop and copy collectors operating over the selected test programs.

<table>
<thead>
<tr>
<th>Test program</th>
<th>Self-scavenging collector for $k = 1$, $k = 150$, $k = 300$</th>
<th>Baker collector for $k = 1$, $k = 150$, $k = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ciresim</td>
<td>39.36 37.04 37.16</td>
<td>45.12 42.87 42.44</td>
</tr>
<tr>
<td>Compress</td>
<td>10.03 09.53 09.42</td>
<td>13.08 12.61 11.95</td>
</tr>
<tr>
<td>Wave4main</td>
<td>12.63 11.79 11.71</td>
<td>16.92 13.63 13.31</td>
</tr>
</tbody>
</table>

Table 9.2: User-times for the (non-profiled) incremental collectors operating over the selected test programs.

- GC_SCAV — Total time spent that is spent collector-scavenging.
- TOTAL — Total execution time.

- Table 9.5 presents the profiling statistics for the Compress test program.
- Table 9.6 presents the profiling statistics for the Wave4main test program.
- Table 9.7 presents the overheads that occur when gathering the profiled collector statistics.
- Table 9.8 presents the average mutator pause times across the test programs when $K$ is varied.

### 9.0.4 Analysis of Results

There are several key observations which can be made about the performance of the collectors:
<table>
<thead>
<tr>
<th>Test program</th>
<th>Number of garbage collection cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ciresim</td>
<td>108</td>
</tr>
<tr>
<td>Compress</td>
<td>501</td>
</tr>
<tr>
<td>Wave\text{main}</td>
<td>136</td>
</tr>
</tbody>
</table>

Table 9.3: Number of garbage collections that occur during execution of the test programs.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>$k = 1$</th>
<th>$k = 150$</th>
<th>$K = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>MUT</td>
<td>12.31</td>
<td>12.69</td>
<td>12.91</td>
</tr>
<tr>
<td>GC</td>
<td>28.11</td>
<td>25.13</td>
<td>24.98</td>
</tr>
<tr>
<td>MUT_SCAV</td>
<td>0.54</td>
<td>0.22</td>
<td>0.12</td>
</tr>
<tr>
<td>GC_SCAV</td>
<td>26.97</td>
<td>24.33</td>
<td>24.27</td>
</tr>
<tr>
<td>TOTAL</td>
<td>40.42</td>
<td>37.82</td>
<td>37.89</td>
</tr>
</tbody>
</table>

Table 9.4: Profiled user times for the \textit{Ciresim} test program.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>$k = 1$</th>
<th>$k = 150$</th>
<th>$K = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>MUT</td>
<td>5.78</td>
<td>5.78</td>
<td>5.06</td>
</tr>
<tr>
<td>GC</td>
<td>7.65</td>
<td>6.74</td>
<td>6.30</td>
</tr>
<tr>
<td>MUT_SCAV</td>
<td>2.67</td>
<td>2.06</td>
<td>1.39</td>
</tr>
<tr>
<td>GC_SCAV</td>
<td>4.88</td>
<td>4.59</td>
<td>4.82</td>
</tr>
<tr>
<td>TOTAL</td>
<td>13.43</td>
<td>12.52</td>
<td>11.36</td>
</tr>
</tbody>
</table>

Table 9.5: Profiled user times for the \textit{Compress} test program.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>$k = 1$</th>
<th>$k = 150$</th>
<th>$K = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INIT</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>MUT</td>
<td>7.67</td>
<td>6.65</td>
<td>6.60</td>
</tr>
<tr>
<td>GC</td>
<td>6.99</td>
<td>5.68</td>
<td>5.49</td>
</tr>
<tr>
<td>MUT_SCAV</td>
<td>0.85</td>
<td>0.40</td>
<td>0.31</td>
</tr>
<tr>
<td>GC_SCAV</td>
<td>5.55</td>
<td>4.66</td>
<td>4.59</td>
</tr>
<tr>
<td>TOTAL</td>
<td>14.66</td>
<td>12.33</td>
<td>12.08</td>
</tr>
</tbody>
</table>

Table 9.6: Profiled user times for the \textit{Wave\text{main}} test program.
<table>
<thead>
<tr>
<th>Test program</th>
<th>Profiling overhead for $k = 1$</th>
<th>Profiling overhead for $k = 150$</th>
<th>Profiling overhead for $k = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cirosim</td>
<td>2.65%</td>
<td>2.06%</td>
<td>1.92%</td>
</tr>
<tr>
<td>Compress</td>
<td>25%</td>
<td>23.96%</td>
<td>17.05%</td>
</tr>
<tr>
<td>Wave4main</td>
<td>13.85%</td>
<td>4.38%</td>
<td>2.66%</td>
</tr>
</tbody>
</table>

Table 9.7: Profiling overheads for the test programs on varying $k$.

<table>
<thead>
<tr>
<th>Average mutator pause time for $k = 1$</th>
<th>Average mutator pause time for $k = 150$</th>
<th>Average mutator pause times for $k = 300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>0.045</td>
<td>0.045</td>
</tr>
</tbody>
</table>

Table 9.8: Average pause time between mutator executions for varying values of $k$.

- The level of accuracy with which execution times can be measured, combined with the variance in execution times for subsequent runs of the same program makes anything but simple analysis of these results difficult. Many of the tables show this.

- It is expected that forcing the use of the slow entry convention and prohibiting access to references in a closure via fast entry points should increase the execution time. For Cirosim and Wave4main, this is indeed the case, but for Compress, this time is reduced. It is possible that these programs do not make much use of fast entry to a closure. In any case, benchmarking against the system that prohibits fast entry is a fairer form of comparison, for fast entry is a form of optimisation, which is currently unavailable to the incremental schemes.

- What should be obvious from Table 9.2 is that for every execution, the self-scavenging collector is more efficient than the Baker collector, i.e. the execution times for the self-scavenging scheme are shorter. The executions of the Baker collector system are between 12% and 25% slower than those of the self-scavenging.
- There is an overhead associated with allowing the mutator to resume before garbage collection is complete. While and Field [While and Field, 1992] suggested that this was around 10% the results suggest an overhead of between 8% and 15%.

- As $k$ is increased, so the overhead of incrementality drops and the execution times are reduced. However, determining $K$, is a very tricky and application dependent task. If $k$ is too low, then mutator progress is rapid and the heap will often be exhausted before scavenging of Tospace is complete. The result of this is that the current garbage collection cycle must be forced to completion and the next one started. In a real-time system, this is highly undesirable for the responsiveness of the mutator, whilst initially appearing to be excellent, degrades rapidly as collections are forced to completion. Tables 9.4, 9.5 and 9.6 show this. The mutator scavenging (MUT_SCAV) times are greater, indicating that the mutator is entering the self-scavenging objects more frequently and thus are being resumed more rapidly. However, in spite of the times between mutator resumption being less in all test cases, every garbage collection cycle is forced to complete with $k$ set to one.

- One might ask why, if the execution time is dropping as $k$ is increased, $k$ cannot be set to a near infinite value. The reason for this, is that increasing $K$ means that less mutator-scavenging is done, and more collector-scavenging is done. As a result a high scavenging limit for $k$ will result in the increased mutator pause time. With an increased pause time, the collector can be classified less and less as incremental. If $k$ is infinite, then all the objects will be evacuated and scavenged before the mutator is resumed. It is therefore a stop and copy collector and is no longer real-time.

- $k$ is clearly dependent on the behaviour of the application and how suited its flow of control is to incremental garbage collection. A program that rapidly allocates objects at a particular stage of computation and then requests little more, is hardly a candidate for incremental collection; constant allocation of new objects is desirable.
• The results show that there is a huge cost associated with the profiling of the collector, especially the collector-scavenge loop and a small value for \( k \) — the scavenger is repeatedly entered and exited very rapidly and as such the overheads associated with starting and stopping the profiler are applied many more times than for a larger \( k \).

• Table 9.8 suggests that for the three programs analysed, the optimum value of \( k \) and the lowest pause time results somewhere between \( k = 150 \) and \( k = 300 \). The average pause time of the mutator provides an indication of the ‘incrementality’ with which garbage collection and mutator execution is being achieved. More correctly however, it is the ratio of the duration of resumed mutator execution to the duration of the pause time that provides this measure.
Chapter 10

Conclusion and Further Work

10.1 Conclusion

Section 1.0.1 stated the following as project objectives:

- ‘To gain an understanding of the issues involved in implementing compilers for functional languages and more specifically the runtime system that supports the compiled program.’

- ‘To evaluate the performance of the While and Field collector through the implementation and benchmarking of a variant of their collector against other algorithms, thus providing further insight into applicable garbage collection techniques for non-strict functional languages.’

- ‘To deliver an incremental garbage collector for GHC 4.01 that supports interactive applications and enforces an upper bound on pause times.’

The first objective was primarily achieved whilst performing the investigative research for the background material of the project, that is presented in section 2 and 3 and also whilst implementing the collectors.

The second objective is split into the implementation of both a self-scavenging collector and a Baker collector and the evaluation of their performance. The implementation of these two collectors was not as trivial
as initially thought, as a result, the time spent benchmarking and evaluating was not long enough to get substantial conclusions. The preliminary conclusions are that:

- Both the self-scavenging and Baker collectors successfully perform (non-optimal) incremental garbage collection but their internal parameters, such as \( k \) are dependent on the allocative nature of the application.

- The programs of the nofib suite, although excellent for the rigorous testing of the compiler and its runtime system, are not really suited to the benchmarking of incremental garbage collectors. Essentially their real-time requirements and interactivity are completely inappropriate. The result of this is that anything but the most basic of evaluations, which has been performed, are impossible.

- The Baker collector has an overhead of between 20\% and 35\% associated with it, whilst the self-scavenging collector has a more favourable overhead of between 8\% and 17\% depending on the application with which it is used. At the lower bound of 8\%, this seemingly supports the preliminary investigations of While and Field who reported an overhead of about 10\%.

The third objective is only partially achieved with a proof of concept for the implementation of an incremental collector into GHC-4.01. There is however, much work to be done on the optimisation of the implementation that has resulted, especially in enforcing a practical upper bound on pause times. In relation to the time allocated for the execution of this project, the delivery of, an incremental garbage collector, based on the While and Field collector, and preliminary benchmark results that required the implementation of a second collector is a challenging feet to accomplish and has been enormously satisfying.
10.2 Further Work

The following section outlines possible ways that the work documented in this report could be extended, including some design ideas for these extensions.

10.2.1 Improving Performance

The time constraints on the implementation of the self-scavenging collector have meant that it is not fully optimised. Two particular areas warrant further investigation when attempting to improve the collector's general performance:

Optimising the Scavenge Limit, $K$

The use of Equation 3.3 results in what appear to be rather low values for $k$. It is suspected, that this equation needs to be modified or perhaps expressed in terms that are more applicable to the storage structures and allocation areas of GHC and the block allocator methods that are used.

No special consideration is taken when incrementing the scavenge count for objects which do not actually result in evacuation of any other objects to Tospaces, such as indirections which are simply followed. It is suspected that a more accurate way of managing the count would be to work in number of words scavenged as opposed to the number of objects. In doing this, the count would no longer be incremented each time round a scavenge loop but in the code that handles the scavenging of each specific type of object. This would therefore reflect more accurately the amount of work that has actually been done when an object is scavenged.

Increase in execution speed

There are two significant ways in which the increase in execution speed of the collector is likely to be achieved:

- Currently, all functions that are associated with closures are forced to enter via their slow entry code. This is not optimal, because various checks namely, heap and stack overflows and argument satisfaction
checks are performed. If the closure is a top-level function which does not require such checks, then a jump to the fast entry point should be allowed. A key extension to this work is therefore the implementation of support for fast entry to top-level functions and the re-benchmarking of the resulting collector.

- The implementation of the two collectors make use of function calls where macros would probably suffice. Although the object code would be enlarged and the source code more obfuscated with the use of macros, the overheads that result from the pushing of activation records and register storage when functions are invoked could be eliminated, resulting in faster execution of the collector.

- The move from a stop and copying collection scheme to an incremental scheme, places extra requirements on the profiling of the collector. A suggested extension would be to devise and implement a more efficient profiler than the one that has been developed.

**Reduction in space overheads**

The benchmark results have shown that the increase in residency that results from the prepending of an info pointer copy word to each evacuated closure is barely significant because the residency values are so low anyway. This appears to support the weak generational hypothesis that 'most objects die young' [Ungar, 1984]. However academically, the design of an algorithm that capitalises on the effectiveness of self-scavenging but eliminates the space overhead required for the extra info pointer copy is desirable.

**10.2.2 Incremental Stack Scavenging**

A possible scheme for incremental stack scavenging that would increase the 'incrementality' and spread collector initiation overheads could be designed as follows:

*Scavenge frames* could be placed at intervals between activation frames in the stack. When the mutator hits one of these extra scavenge frames, the scavenge code is executed to scavenge that section of the stack up to
10.2 Further Work

the next scavenge frame. This first scavenge frame would then be removed
from the stack. An efficient way of implementing this would be to traverse
the stack at initiation of the collector and mark some of the update frames
as also being scavenge frames. Obviously, the initial part of the stack up
to the first update frame would have to be scavenged at this time. When
the mutator hits one of these marked update frames, the scavenge code is
executed and then the update is performed.

This scheme could prove to be no more efficient than scavenging the en-
tire stack at garbage collector invocation. The update frames on the stack
are not distributed evenly throughout it, i.e. they are clumped together.
This means that the marked update frames (scavenge frames) will not nec-
essarily be evenly spaced and at each scavenge, a different amount of work
may be done resulting in the user seeing more uneven pause times. It is al-
most certain that the mutator will not have invoked all the scavenge/update
frames by the end of garbage collection and so the collector must do this
itself. Depending on how many frames have been processed, this amount
of work may be close to that of scavenging the entire stack in one go at
collector invocation.

PAPs (Partial Applications) contain small chunks of stack, as a result,
they could also use this scavenging scheme although whether the overhead
of initiating it for such a relatively small object is questionable.

10.2.3 Incremental Scavenging of Unpointed Objects

The following subsection outlines possible schemes for the incremental scaven-
enging of some of the unpointed objects, thereby increase the ‘incremental-
tality’ of the collector. The term ‘lazy’ is used to imply the occurrence of
scavenging (often complete) at a later stage to evacuation.

Unpointed Objects Accessed via Primitives

For objects that are accessed via primitive operators that use offsets/index
into the object, the code for these primitive access operators are modified
to include a read-barrier. Examples of such operators and objects are the
array read/index routines. The only way that a Fromspace object may be
accessed is via primitive read operators and so write operations need not be touched.

**Lazy Scavenging of Foreign Objects**

An alternative strategy to the immediate evacuation and scavenging of foreign objects, involves the implementation of a read-barrier. There are no primitives that may be used to access a foreign object. The only way in which the contents of the object is retrieved is to pass it as an argument to a \_c\_call\_\_ The compiler code could thus be modified to implement a barrier here.

**Lazy Scavenging of TSOs**

TSO objects could possibly be scavenged lazily by amending the code that executes the thread with a read-barrier or self-scavenging code resulting in the scavenging of the TSO before the thread runs. This could be a very effective scheme when combined with the stack scavenging mechanism described above.

**Lazy Scavenging of Stable Pointer Objects**

The only way for the mutator to access an object pointed to by a stable pointer is by calling \_d\_e\_r\_e\_f\_S\_t\_a\_b\_l\_e\_P\_t\_r\_()\_, hence, it should be possible to place a read-barrier around this code.

**10.2.4 Implementation of an Incremental Collector into GHC-4.02**

The performance figures of the self-scavenging collector are very promising and suggest that further development of some form of incremental collector and possibly even further development and optimisation of the self-scavenging collector could potentially provide the latest version of GHC with the means to provide a more seamless flow of execution for interactive applications. The results also indicate that there is possible scope for the development of a hybrid collector that varies the algorithm used depending both on the state of the heap and how interactive an application is. How the
classification of the ‘interactivity’ of an application may be made has not as yet been researched fully, but possible heuristics could be used that take into account the number of I/O calls made in the program. The collector could also be invoked after a certain pause time which occurs while the mutator is blocked waiting for user input.

A Full While and Field Collector Implementation

Further investigations could be carried out to determine precise figures for the degradation that the performance of the compiler and runtime system experience when code pointers are placed in the info table and the fragmented scavenging routines are used. Comparison of these overheads with those of the self-scavenging collector and the stop and copy collector could be used to evaluate the worth of performing the full implementation of a While and Field collector into GHC.

Over the implementation phase of the collectors that support this report, the GHC runtime system developers have implemented a new generational collector into GHC-4.02. A further investigation, that would complement the work presented in this report, would be a study of the applicability of the While and Field scheme to generational garbage collection techniques. The practical investigation could be performed on GHC-4.02’s generational collector.

Enhancing the Baker Collector

It has been demonstrated that the implementation of Baker’s algorithm is not as efficient as the self-scavenging scheme in terms of the overheads placed on execution speeds of the user program. The implementation can however be enhanced to eliminate the space overhead that is used to store a copy of the info pointer. The primary purpose of this word in the self-scavenging collector is to maintain a copy of the original info pointer which is restored when the object has been scavenged. A most useful ‘side-effect’ of this design is that this word can be used to resolve the update problem of closures being overwritten by smaller objects, thus leaving holes in Tospace. These holes makes it difficult for the scavenger to find the next valid closure. In [Peyton
Jones, Marlow, Reid, 1999] the concept of a slop object is introduced. This object is recognisable by the scavenger and indicates to the scavenger of the number of slop words that must be skipped to reach the next valid closure. The use of slop objects in the Baker collector would eliminate the space overhead mentioned above.
Bibliography


Conclusion and Further Work


10.2 Further Work


