Typed Interactive Memory Management
Master's thesis
Department of Computer Science at the University of Copenhagen (DIKU)

Ánoq of the Sun, Hardcore Processing [1]

August 12, 2010

Quoted as e.g.: Ánoq of the Sun, Ánoq, o., Ánoq, o. t. S. or Ánoq, of the Sun. Not: Sun, Á. Ánoq is considered the "family name", always written and pronounced first.

Contents

Resumé på dansk

Dette speciale beskriver avancerede dynamiske metoder, til at håndtere lagerstyring i computerprogrammer, specielt til brug i oversættere. Fokus vil blive lagt på at præsentere en stor mangfoldighed af metoder, som især ville kunne bruges til programmer af interaktiv natur, såsom computerspil eller andre programmer, som kræver, at programmerne ikke pludseligt går i stå et øjeblik, for at skulle rydde op i lageret. Et andet fokus i rapporten er, at vise, hvordan typer i programmeringssprog kan bruges, til at opnå nogle optimeringer eller andre fordele ved implementering, som ikke nødvendigvis kan opnås på andre måder.

Mange metoder bliver gennemgået. Gennemgangen er både på det generelle plan, for at give et overblik over, hvordan det er muligt at sammensætte, udvide og ændre allerede eksisterende metoder. Specifikke metoder, som tidligere er beskrevet andre steder, vil også blive præsenteret i stort omfang. Både statiske og dynamiske metoder gennemgås og det fremgår, at det som oftest kun er de dynamiske metoder, som i praksis giver en komplet løsning til at styre dynamisk hukommelsesforbrug.

Præsentationen af generelle og specifikke metoder munder ud i en kort analyse, som foreslår nogle metoder og kombinationer af metoder, som kunne vaere gode at vælge, hvis man skulle lave et hukommelsesstyringssystem, som kan bruges til både interaktive og generelle programmer.

En nyligt publiceret metode, *mark-region*, bliver foreslået og gennemgået til implementering. Mange detaljer og aspekter ved implementeringen bliver præsenteret og de fleste dele af implementeringen bliver enten vist, forklaret eller gjort tilgængelige. Dele af implementeringen, som kræver yderligere arbejde, bliver også analyseret og forslag til udvidelser bliver givet.

Det implementerede system bliver evalueret med praktiske målinger, som viser, at systemet virker som man kunne forvente, i forhold til det implementerede og de valgte metoder.

Nye bidrag fra projektet inkluderer en moderne implementering af brug af statiske typer, til at gennemløbe det dynamisk styrede lager, tilpasset den nyligt udgivne mark-region metode. Ventetider ved lagerstyringen er også målt, i modsætning til i den udgivne artikel om metoden. Endeligt præsenteres der undervejs eksempler og grænseflader til programmoduler, som er starten på at gøre det muligt, skrive hukommelsesstyringssystemer til oversættere i det kildesprog, som oversætteren selv oversætter fra.

Dele af denne afhandling blev hastigt lavet i sidste øjeblik. Jeg håber, at læseren bærer over med mig på dette punkt. En fejlkorrigeret version vil blive udarbejdet efter specialeforsvaret.

Chapter 1

Introduction

Many modern programming languages require automatic memory management. There are several good reasons for this. It frees the programmer from having to worry about the details of freeing memory. It also enhances programmers' chances of understanding a program's local modules, without having to worry about how its memory is managed when used by other modules. For languages with explicit memory management, if memory is managed incorrectly, it even becomes a security concern: either the memory leaks or references to unallocated or wrongly freed data occurs. There exist debugging tools for this, but they tackle the symptoms rather than the disease itself, as also stated in section 1.4 on page 10 in [Jone96]. As computer programs grow larger, all of these considerations become increasingly important. Automatic memory management is however not free of challenges or problems of its own, but hopefully its usefulness is evident.

This thesis gives a general overview of some commonly used methods and paradigms for automatic memory management. It also explores more recent and specialized methods, with the aim of finding relevant methods for implementation. The desired goals for the implementation is to provide the basis for a complete memory-management system, which is suitable for interactive programs when used with a compiler for a functional programming language in the spirit of Standard ML. Another goal of the thesis is to show how to take advantage of static type information from a type-preserving compiler, in order to improve on the memory-management system.

I got a personal motivation for making a memory-management system suitable for interactive programs when I wrote two small computer games in Standard ML back in 2001. When trying to compile one of the games with the compiler ML Kit [MLKit], it ran out of memory, since I was not able to figure out how to make the special kind of program modifications required to avoid memory leaks when using region inference. One of the promises of region inference is that it can avoid garbage collection pauses, but trading pauses for memory leaks is a serious problem. When compiling the games with the compiler MLton [MLton], as done in the released versions of the games, they exhibit occasional garbage collection pauses, which is particularly evident in one of the games. The general conclusion here is that memory management suitable for interactive applications is necessary, if programming languages like Standard ML are to be used for game development or for performing time-critical tasks.

1.1 Basic Terminology and Definitions

The reader is expected to have some basic understanding of computer science. This includes some understanding of algorithms, time and space complexities and data structures, particularly trees and graphs connected by pointers in memory, where terms like children and child pointers are assumed to be

well-known. It is also expected that the reader is familiar with the basics of how computers and operating systems work, particularly concepts like cache, memory pages and memory protection, virtual memory, threads, thread-locking and context switches, although we shall not dive deeply into such issues. The reader is also expected to know the basics of how a compiler is constructed, since the thesis describes a memory-management system intended for incorporation into an existing compiler.

Some of the basic terminology and definitions used in this thesis are listed here:

1. **The heap**: Normally the largest portion of memory in a program or a computer. Memory may be allocated and freed from this in any order, with respect to the program execution

2. **The stack**: This normally refers to the call stack of a program, from which memory may be allocated according to the last-in-first-out principle, following the order of function call invocations

3. **A (memory) node, cell, area or object**: These terms are used interchangeably (also defined in section 1.2 page 4 in [Jone96]) and refer to individually allocated pieces of data in the *heap*

4. **Heap manager, external and internal**: The heap manager manages the allocation and deallocation of memory areas. The *external* heap manager is usually the operating system and is assumed to somehow be capable of allocating and freeing memory areas. The *internal* heap manager is the data structure used to manage the memory areas allocated by the external heap manager, usually for the more fine-grained allocation and deallocation required by the program (mutator) and the rest of the memory management system (e.g. the garbage collector). Since the program and the memory management system usually interact with the internal heap manager, the *internal* heap manager is what is meant when specifying just heap manager

5. **Allocated memory**: Memory areas referred to as being *allocated* are allocated by the heap manager, from where they are returned as valid usable memory areas. This may be specified independently for both the external and the internal heap managers. *Unallocated* memory is the opposite of allocated, but also includes *deallocated* memory, i.e. memory which had been allocated at some point during program execution and then freed again. Memory may also be allocated on the stack, which will be referred to as *stack allocation*

6. **Free memory**: Memory areas referred to as being *free* are either unallocated by the heap manager, or have been freed to the heap manager, which manages the memory areas as being free. This may be specified independently for both the external and the internal heap managers. We normally are not concerned with how the external heap manager manages free memory, but in some contexts we are concerned with how the *internal* heap manager manages it explictly. It should be noted that memory allocated by the external heap manager, but maintained as being free by the internal memory manager, is normally considered free memory, unless otherwise stated. Memory may also be freed from the stack, which will be referred to as *stack deallocation* or *stack freeing*, normally accomplished by just moving the stack pointer, which defines the top of the stack, but sometimes this may even be done implicitly, without any operations, in case the stack pointer points to the top-most function stack frame

7. **Roots or root set**: The combined set of the values on the program stack (local variables and temporaries), values in registers and global values (also defined page 4 section 1.2 in [Jone96])

8. **Live (memory) object**: A memory object is *live* if its address is held in the root or there is a pointer to it from another live memory object (also defined page 4 section 1.2 in [Jone96])

9. **Liveness**: *Liveness* of the graph of memory objects (nodes) in the heap is defined by *pointer reachability* from the root set (also defined in section 1.2 page 4 in [Jone96]). Notice that this is a *conservative estimate* of liveness, since e.g. local variables could be dead (i.e. no more used) or stack frame slots could be uninitialized, as also explained on page 5 in section 1.2 in [Jone96]

10. **Direct and indirect liveness determination**: Liveness may be determined *directly*, which is usually done by *reference counting*, or it may be determined *indirectly*, usually by *pointer tracing* from the root set

11. **Garbage**: Memory objects are *garbage* if they are not live but not free either, i.e. allocated but not (or no longer) used (also defined page 5 in section 1.3 in [Jone96])

12. **Dangling (pointer) reference**: A (pointer) reference is *dangling* if it points to an unallocated (possibly deallocated) memory object (also defined on page 6 in section 1.3 in [Jone96])

13. **Pointer sharing**: When more than one pointer points to a particular memory area, that memory area is *shared* among all the pointers which point to it from live memory areas (also defined on page 7 in section 1.3 in [Jone96])

14. **The mutator or the (user) program**: The user program is called, interchangeably, the *mutator* or the *(user) program*, since it changes or mutates memory objects and the graph that connects them (page 2 section 1 in [Jone96]). For a multi-threaded program, each thread is a mutator

15. **The (garbage) collector**: The system and algorithm which manages dynamically allocated storage. This thesis is primarily about the methods and algorithms of *(garbage) collectors*

16. **A collection and a collection cycle**: A *collection* (when not referring to a well-known data structure with the same name) is the process of performing garbage collection for the heap. For incremental algorithms, this may be done in steps. In incremental settings, a *collection cycle* is the process of performing all incremental steps of garbage collection until the targeted areas of the heap have been garbage collected. This is independent of whether the collection performed targets the full heap or only a part of the heap (typically used for generational garbage collectors)

17. **Resident (or occupied) memory**: The ratio $r = R/M$ of the amount of live memory R to the size of the heap M (or actually, the amount of memory allocated for the memory manager). The formula is from section 2.4 pages 34-35 in [Jone96]

18. **The memory management system**: The *memory management system* comprises both the heap manager and the collector and any other parts required by these

19. **Incremental memory management**: When a memory management method is incremental, it can perform the memory management in small steps. This consideration is independent of whether multiple concurrent threads are involved or not

20. **Concurrent memory management**: When the memory manager (typically a collector) runs concurrently with the mutator. This is regardless of whether the mutator is a single-threaded or a multi-threaded application

21. **Multi-thread support**: A memory management system may or may not support multi-threaded applications, where each thread is considered a mutator. This consideration is independent of whether the collector is concurrent with the mutator(s) or not

22. **Interactive memory management**: This means that the memory management is done in such a way that the pauses for performing garbage collection are sufficiently short that mutators can run at interactive rates without too noticable disturbances. A commonly used phrase, also cited in [Jone96] on page 144 in section 7.1, is: "Can I garbage collect while tracing the mouse?"

23. **Real-time memory management**: Many, especially earlier, concurrent or incremental algorithms are described as being "real-time" in the litterature. Classifying them as interactive would likely be more appropriate. For an algorithm to qualify as *real-time*, it normally must meet some well specified bounds on pause times, which may vary depending on whether e.g. *soft real-time* or *hard real-time* is meant (page 223 section 8.10 in [Jone96])

The reader is expected to have a good understanding of these terms. The terms are sufficiently basic that we shall not refer to these definitions, yet sufficiently important that some of the distinctions between the terms and the contexts within which they are used are crucial for understanding the thesis.

1.2 An Introduction to the Common Garbage Collection Methods

The most basic and well-known methods for memory management are *garbage collection* methods. A large part of this thesis is about such methods. As we will see, other methods considered are typically not full solutions to dynamic memory management in themselves, which is why the primary implementation and analysis effort will be centered around garbage-collection methods. The following sections briefly present the most basic and well-known garbage-collection methods.

1.2.1 Reference Counting

Reference counting is a very well-known method for memory management. It is an explicit method, where the number of references to each memory cell is kept track of explicitly in a reference counter. The reference counter is increased whenever an additional reference to the memory cells is created. As long as the reference counter for a memory cell is above zero, that memory area is live and should not be freed. When the counter reaches zero, i.e. when the last reference to it is freed, the memory cell may be freed. Whenever a memory cell is freed, its children (if any) that it points to must have their reference counters decreased.

Reference counting has been used directly by developers for keeping track of when it is time to free objects in memory, which in this case may refer to programmer-level objects, not just memory objects, as in the rest of this thesis. Every time a part of the program uses an object, it increases the reference counter, which it decreases again when done using the object. The object is thus "pseudo-automatically" freed when it is no longer needed. This gives some challenges with returning objects from functions, which in the OpenStep API (now Cocoa, as used in Apple's Mac OS X; see e.g. [GNUStep] for information on these APIs) handles by autorelease pools and delayed freeing of objects. This method is also still subject to the problems of memory leaks and reference to no longer available memory, since the programmer is required to count the reference up and down correctly.

When referring to reference counting in this thesis, it is not the programmer-managed reference counting seen in the OpenStep API which is meant, but rather a method automatically managed by a compiler. The goal is to make memory management fully automatic, which requires the compiler to do the work automatically. For reference counting, the references can be managed by the compiler by properly counting references at the points of allocation of memory and by determining when references to objects die, by performing liveness analysis, which is also commonly performed for register allocation in a compiler.

1.2.2 Copy Collection

The copy-collection method is one of the two primary *tracing* collector methods. It is therefore an *implicit* memory-management method. It divides the heap into two *semi-spaces*. One semi-space, *Tospace*, contains the current active memory while the other semi-space, *Fromspace*, contains the obsolete data. When garbage collection is performed, the roles of the two spaces are first *flipped*. The collector then traverses the active data structure in the previously active semi-space, now Fromspace, copying each live cell into the new active semi-space, now Tospace.

The basic canonical copy collector method is known as Cheney's algorithm. Cheney's algorithm is presented on pages 118-123 in section 6.1 in [Jone96] and in section 13.3 in [Appe98]. The algorithm uses just two pointers of memory, `a` (also sometimes referred to as `free`) and `s` (also sometimes referred to as `scan`), to represent two end points of a traversal queue. It uses space in the scanned (grey) memory cells in Tospace, after they have been copied, for storing the traversal queue. If this was not done, copy collection would require extra stack space just to perform the garbage collection recursively. Therefore, copy collection algorithms normally means Cheney's algorithm or better nowadays.

An implementation of most of the method is shown in Standard ML in figure 1.1. The algorithm starts by flipping the roles of Tospace and Fromspace and by initializing `a` to point to the bottom of the (new) Tospace and then copying the root set on the program's (or thread's) call stack into Tospace, advancing the allocation pointer `a` as objects are copied. This initial part is not shown in the code in figure 1.1. `s` is then initialized to point to the bottom of Tospace. At each iteration, the next grey cell (pointed to by `s`) is scanned for child pointers to objects in Fromspace that have not yet been copied. Whenever an object in Fromspace is encountered, it is evacuated to Tospace and a forwarding address is left behind in Fromspace. Some means is necessary to distinguish pointers from non-pointers, either statically or dynamically by using e.g. a header descriptor, letting heap objects be segregated by their type or tagging to distinguish pointer words from non-pointer words. The implementation in figure 1.1 assumes tagging for the distinction between forwarding pointers and values to be copied, which could also be used to distinguish objects, although this might still require a few changes to the implementation. Determination of sizes and child-pointers of scanned objects is left unimplemented in the figure.

Notice that except for the unspecified computation of `valueSize` and `offsets`, the algorithm in figure 1.1 is actual Standard ML code, although it has not been tested or compiled in this project. The article [Hall02] uses Standard ML-like code, but this is somewhat heavily overloaded pseudo-code. It is a contribution of this thesis to define Standard ML modules which allow implementation of such an algorithm. Key to this is the specification of a module called `UnsafeMemoryAccess`, which will be loosely specified later in section 8.3, particularly in the figure 8.4. The functions `UnsafeMemoryAccess.addrtoword` and `UnsafeMemoryAccess.wordtoaddr` convert between the opaque address type `addr` and a type `WordAddr.word`. The module `WordAddr` is either the same as `Word32` or as `Word64`, depending on architecture, but it should be defined by structure assignment, for portability of code. The functions `UnsafeMemoryAccess.readaddr` and `UnsafeMemoryAccess.writeaddr` read, respectively write, an address value at a given address with an offset.

1.2.3 Mark-Sweep and its Relatives

The mark-sweep method is the other of the two primary tracing collector methods and is therefore, like copy collection, also an implicit memory management method. It consists of two main phases:

1. Mark all memory cells reachable from all roots, usually with a mark bit allocated for each memory cell or with some bitmap of the memory layout. This may require clearing all mark bits first

```
(* Copy a memory word from src to dest addresses, both with offset *)
fun copyWordWithOffset(src : addr, dest : addr, offset : int) =
      UnsafeMemoryAccess.write32(dest, offset, UnsafeMemoryAccess.read32(src, offset))

(* a is the (updatable) allocation pointer in toSpace and
   p is the pointer in fromSpace to evacuate from.
   valueSize is the number of words occupied by the value to evacuate *)
fun evacuate(a : addr ref, p : addr, valueSize : int) =
      let
        val opaque = UnsafeMemoryAccess.readaddr(p, 0)
        val value  = UnsafeMemoryAccess.addrtoword(opaque)

        val modifiedValue =
              if WordAddr.andb(value, 0wx1) = 0wx1 then
                (* The word at p is tagged, assumed to mean a value *)
                let
                  (* Remember the address that we evacuate to, then
                     advance allocation pointer *)
                  val a0 = (!a)
                  val () = a := UnsafeMemoryAddress.addradd(a0, valueSize)

                  (* Copy the evacuated value from fromSpace to allocation ptr. *)
                  val () = Int.forl(0, Word.fromInt valueSize, 1, fn (i, ()) => copyWordWithOffset (p, a0, i), ())

                  (* Install forwarding (i.e. unmarked) pointer in fromSpace *)
                  val () = UnsafeMemoryAccess.writeaddr(p, 0, a0)
                in a0 end
              else
                (* The word at p is untagged, assumed to mean a forward ptr. *)
                value
      in
        UnsafeMemoryAccess.wordtoaddr(modifiedValue)
      end

fun scanValue (a : addr ref, s : addr ref) =
      let
        val valueSize = (* Size of value stored at !s *)
        val offsets = (* List of child-pointer-offsets in value stored at !s *)

        (* Evacuate children of the value stored at !s *)
        val () = List.app
                   (fn offset =>
                      UnsafeMemoryAccess.writeaddr(slot, offset, evacuate(a, slot, valueSize))
                   )
                   offsets
      in
        (* Advance scan pointer to next memory object *)
        s := UnsafeMemoryAddress.addradd(!s, valueSize)
      end

(* The root-set is assumed to have been placed in toSpace
   prior to this, ending at the allocation pointer address a *)
fun cheney(fromSpace : addr, toSpace : addr, a : addr ref) =
      let val s = ref toSpace   in while (!s) < (!a) do scanValue(a, s) end
```

Figure 1.1: Source code in Standard ML for Cheney's copy collector. This has not been tested or compiled in this project, but unlike the pseudo-code presented in section 2.2 in [Hall02], this is actual code, except for the two first lines in the function *scanValue*, which could either be determined statically or dynamically by object headers (in which case the algorithm might have to be modified accordingly). The deeper meaning of certain details of the *evacuate* function will be investigated further later on. A loose specification of the module *UnsafeMemoryAccess* can be found later in section 8.3, particularly in the figure 8.4. The function *Int.forl* is from [AMLB09]; it iterates *valueSize* iterations, starting at zero and advancing one at each iteration, where the value *i* is the iteration counter in the given function

2. Sweep the entire heap, e.g. linearly, where all unmarked memory cells are freed

The first of these two phases is effectively a direct implementation of computing the liveness of all memory cells. The second phase reclaims all garbage, i.e. unused memory objects.

There is a related method to mark-sweep, known as mark-compact. In this method, live objects are copied to one end of the heap during the sweep-phase, which can eliminate problems with fragmentation, at the cost of copying more data.

1.2.4 Common Limitations of Simple Garbage Collection Methods

When considering the simple garbage collection methods presented here, at least in their plain, canonical and unimproved forms, they have several limitations or inefficiencies. These include: 1) high overhead (either in space or processing cost), 2) pauses during collection, fragmentation of memory, 3) bad cache (locality of reference) or paging behaviour and 4) uneven mutator utilization. The problem of uneven mutator utilization is slightly more high-level in nature and refers to when the pause times during collection are kept low, but they occur with such unevenly distributed time intervals as to make the mutator almost unable to perform computations in certain periods of the execution, typically when the garbage collector needs to frequently collect garbage.

This thesis presents several advanced garbage-collection methods which are able to overcome these limitations and sources of inefficiency. Most methods, however, are not able to overcome all of the limitations.

1.3 The Goals of this Thesis

Some of the desired properties of the memory management method to be developed in this thesis are the following, ordered as much as possible according to preference:

1. **General purpose memory management for a compiler**: We seek a general purpose memory management method, suitable for use in a compiler. Thus, in this setting, the compiled application is the mutator, as are possibly many of the compiler's basis library modules

2. **Interactive applications, short pause times**: The method should be suitable for interactive applications. This means that there should not be any long pauses. It is not a goal to meet any particular real-time constraints, but the closer it comes to this, the better

3. **High performance**: It should preferably be efficient. It is likely necessary to sacrifice some performance, in order to achieve the previous goal above, but the less is sacrificed here, the better

4. **Multi-thread support**: The method should preferably scale to multi-threaded applications and to run on symmetric multi processor (SMP) computers. The reason for this goal is that this is becoming increasingly important with modern computers. However, implementation of this will not be considered, since e.g. according to [Auer07], it took a significant amount of engineering work to scale their system to this, so it is outside the scope of this thesis. Also, the compiler used for hosting the memory management system does currently not support any kind of concurrency

5. **Static type information**: Showing or arguing how to use types for improving the methods is an important topic of the thesis. However, full advantage of this will not necessarily be taken in the implementation, where only relatively simple methods related to types will be used

6. **Low space overhead would be nice, but not crucial**: Space overhead should ideally be small, but since both achieving high performance and short pause times often requires extra space, space overhead is the most accepted compromise to make, out of these three performance criteria

7. **Cache and paging behaviour**: While cache and paging behaviour is very important for the performance a memory management system, improving on this will not be an important goal in this thesis. Also, the effects of cache and paging will not be measured in any way. However, when there is a choice of methods, their potential in this respect is a relevant consideration

8. **Compiler cooperation is acceptable**: It is acceptable that cooperation from the compiler is necessary for the memory management system. It would be necessary anyway, if we are to take advantage of static type information. Avoiding necessary cooperation from the compiler would be relevant, if we were building a memory management system for an existing given compiler, without any options for modifying it, but since the aim is to extend a compiler with a memory management system, such required cooperation is not a problem

9. **Binary interface considerations**: It is relevant that the memory-management method is suitable for compiling stand-alone libraries, such as dynamic link libraries (DLLs). I will however only go through some hints of possible ways of achieving this, but it will not be implemented

10. **Functional programming language**: The memory management system is intended for use with a compiler for a Standard ML-like functional programming language, CeXL, meaning that some assumptions about object demographics can be made, such as many small and temporary objects being created and only objects of certain types, e.g. references and arrays, are ever updated. It would however be preferable if the memory manager did not rely too much about these assumptions, in order to perform well. Avoiding such assumptions might, for instance, preserve the performance when reusing the same compiler back-end for other source programming languages

11. **Few assumptions about applications**: One intention of the memory management system is that it is suitable for interactive applications like 3D computer games. Applications like 3D games could be seen as having certain behaviours, like it being advantageous to do garbage after every rendered frame and the game having a substantial amount of permanently live data, such as its 3D world. However, it is preferable that as few assumptions as possible are made about the application's memory requirement patterns when considering the choice of methods

Chapter 2

Previous Work

[Jone96] is considered a good and authoritative reference on the subject of dynamic memory management. Chapter 13 in [Appe98] is also a good reference. Chapter 7 in [AhoL07] also contains an overview of the subject. Often, systems claiming to be real-time do not actually meet any hard real-time constraints and should therefore probably be referred to as being just interactive. Some of the best attempts at making software-only real-time garbage collectors until around 1996 (according to section 8.9 in [Jone96]) were Nettle and O'Toole's system [Nett93] and Baker's Treadmill [Bake92]. Since then, other systems have also become relevant. [Baco03] presents a real-time uni-processor garbage collector with low overhead and consistent utilization. The article [Auer07] presents the implementation of a complete Java Virtual Machine (JVM) product, including compilation, memory management and execution of programs. They use an extended version of the garbage collector from [Baco03], which they call Metronome. They have extended it from the uni-processor system not supporting all Java's RTSJ features, as presented in [Baco03], into being a symmetric multi processor (SMP) system supporting all RTSJ features and having the worst-case latencies reduced by an order of magnitude. The system in [Auer07] thus achieves real-time garbage collection with sub-millisecond worst-case latencies. The system from [Baco03] and [Auer07] will be reviewed in detail later in this thesis, in section 6.7.

[Blac03b] presents a hybrid garbage collection scheme, using a copying collector for the young memory objects and reference counting for the old objects, thus using the strengths of both of these methods to compensate for their individual weaknesses.

[Boya03] uses region-based memory-management on real-time threads for Java, but this does not include all threads in a program and seems to require a bit of special programming.

[Blac04a] presents a modular toolkit for garbage collection for Java, called MMTk. They state that it is modular and composable, yet efficient, and that their approach shows that performance-critical software can embrace high-level languages and modular design.

[Dhur03] and [Dhur05] develop a type-system for C, which ensures memory safety for a large subset of type-safe C programs, where their focus is on embedded C programs. This gives memory safety with explicit allocation and deallocation of memory, as done in C, and is thus an example of not using automated dynamic memory management.

[BenY02] extend an existing mark-sweep method from supporting only full-heap compaction, into also using an additional incremental and concurrent compaction, in order to decrease the maximum pause times without too much loss of performance.

[Blac08] presents a mark-region garbage collector and combine this with what they call opportunistic defragmentation. They get all the advantages of the canonical tracing collectors: mark-sweep, mark-compact and semi-space copy collection. This article does, however, not address pause times. A very

recent article [Pizl10] builds onto their method and focus on concurrent real-time applications with hard real-time constraints.

[Chea04] presents an incremental garbage collector for the Haskell programming language. Their design is intended for production use with the Haskell compiler GHC [GHC] and improves on earlier work of theirs. They use code specialization to achieve some of the performance improvements and show how the same techniques can be used to eliminate write barriers in a generational garbage collector. Their methods are, however, fairly tailored to the implementation of functional programming languages with lazy evaluation, such as Haskell, which is quite different from languages with strict evaluation, such as Standard ML and most imperative languages today.

The article [Chen06] describes dynamic profiling and optimization to garbage collection. They achive better cache locality and page locality at a low cost in overhead of dynamic sampling. This is not a garbage collection strategy in itself, but an improvement, which can probably be made to many garbage collectors.

[Vech06] presents a framework for deriving and synthesizing concurrent garbage collection algorithms, which are proven correct. This provides many insights about the nature of concurrent collection and lays the groundwork for the automated synthesis of correct concurrent collectors.

[Mann05] presents an interprocedural static data flow program analysis, which is able to conservatively bound an application's allocation rate. This is an important step towards meeting hard real-time constraints on garbage collection.

The article [Demm90] presents some interesting theories. Particularly a theory which can transform easily computable reachability algorithms into being as precise as desired. The article also presents two concrete garbage collector methods, one of them being the sticky-mark-bits algorithm, which allows forming in-place generational garbage collection from non-generational methods. This will be reviewed in more detail in section 6.10.

As a disclaimer, I have only read the abstracts of the articles [Boya03], [Dhur03], [Dhur05], [Cors03], [Chen06], [Vech06] and therefore cannot vouch for whether or not they actually provide the results that their abstracts seem to promise, nor will these methods be presented in any more detail than here.

Chapter 3

Basic Considerations for Memory Management

This section gives some general and basic considerations regarding memory management.

3.1 Typical Overhead, which is Hard to Quantify

In broad and general terms, the following can be stated about the overhead of memory management:

- **The cost of memory management is hard to quantify:** A quote from section 1.5 page 13 in [Jone96]: "The cost of automatic memory management is highly application- and language-dependent so it is not possible to give simple prescriptions for its overhead."
- **Typical overhead:** At the end of section 1.5 page 13 in [Jone96], it is stated that the overhead of garbage collection usually ranges from a few percent to around 20% and that 10% is not unreasonable for a well-implemented system

This basically means that, there is neither going to be any "easy nor free lunch".

3.2 Global and Stack Allocation

Certain older programming languages allowed only global memory allocation. While completely free of runtime overhead, it would put a very heavy burden on the programmer to manage this for modern applications. Some values, especially constants, may still be allocated globally, which is of course desirable and should be done.

Allocating memory on a stack is less restrictive than global allocation and is very simple and efficient at runtime. The following can be said about stack allocation:

- **Stack allocation has its limitations:** General dynamic memory allocation is not possible with stack allocation, which is why more refined memory management methods are normally needed (see section 1.1 in [Jone96])
- **Stack allocation is desirable:** Stack allocation and deallocation is very simple and efficient and has very good cache behaviour (locality of reference), which is why it is generally preferred to be used for as much of the allocated memory as possible

- **Stack allocation in performance measurements:** When making experiments of evaluating other memory management strategies than stack allocation, one has to consider whether it is used in combination with stack allocation where possible or not. Avoiding stack allocation may put more stress on the general memory management system and therefore, in some way, be a better test of its capabilities. On the other hand, stack allocation should be done as much as possible in a real-life implementation, so not combining with stack allocation may make the experimental results less useful in practice. The experiments could therefore be run both with and without stack allocation, although there will not be time to do experiments with that kind of thoroughness. There are, however, varying degrees of how much stack allocation is done: better program analyses and clever translation can improve the amount of stack allocation possible, so results obtained with stack allocation enabled still requires care for real-life considerations

- **Storing the stack in the heap:** Moving to the opposite extreme of trying to maximize stack allocation, it is possible to store activation records of function calls on the heap, although they are normally stored on the call stack. The advantages are that there is only one root to trace and that one does not have to worry about stack-overflow. It also implies that when performing generational garbage collection (as will be introduced later), it is easier to trace only the newly allocated parts of the heap. Further, using continuation-passing style (CPS) in the compiler's intermediate languages and supporting the "call with current continuation function" (call/cc) becomes natural. The disadvantages are more frequent garbage collection and the need for having an explicit pointer to the activation record of the callee of a function. It may also result in poorer locality of reference, although this may depend on the garbage collection method; as it turns out, the method that we end up implementing should have very good locality of reference, so allocating stack and heap-space simultaneously might be advantageous in such a case. Allocating activation records on the heap will, however, not be considered further in this thesis, since the compiler for which it is to be incorporated currently uses stack allocation

3.3 Object Demographies

The following has been noticed by researchers about the lifetimes, layouts, creation and modification patterns of memory objects:

1. **Many memory cells are short-lived:** This is stated in section 2.1 page 22 at the bottom in [Jone96] and stated as the *weak generational hypothesis* on page 144 in section 7.1 in [Jone96]. Something similar is also stated as the generational hypothesis in section 1 in [Blac03b]: young objects mutate (i.e. are updated or created) frequently (with reference to [Appe89] and [Stef99]) and die at a high rate (with reference to [Leib83] and [Unga84]). Section 7.1 on pages 142-146 in [Jone96] has several references showing that most objects are short-lived

2. **Old objects mutate (i.e. are modified or created) infrequently and die at a slower rate**: This is stated as the *strong generational hypothesis* on page 146 in section 7.1 in [Jone96]. This is also stated in section 1 in [Blac03b] with reference to [Appe89]. It is also observed, according to the measurements section 4.5 in [Blac03b]. These measurements are, however, for Java and the SPEC JVM benchmarks, so they could potentially be specific to imperative or object-oriented programming languages or specific to that set of benchmarks. However, it seems reasonable that this is a general behaviour for memory objects

3. **Memory objects are typically allocated and freed in clusters**: According to a reference in section 6.6 page 131 in [Jone96], it has been observed that over 60% of the longest lived objects were allocated within one kilobyte of each other, where this correclation is even stronger if younger objects were considered. These objects also died in clusters, which strongly suggests that the initial layout of heap objects reflect the future access patterns by the user program. This was confirmed by four other references (see page 131 in [Jone96]) for the languages Lisp and Smalltalk as well

4. **Few memory cells are shared:** This is at least true for Lisp and functional programming languages. It is stated at the bottom of page 22 in section 2.1 and at the bottom of page 81 in section 4.2 in [Jone96], where three references are given for this

3.4 Programming Language Considerations

- **Functional versus imperative languages**: Functional languages, like Standard ML and CeXL, use mostly immutable data and typically has many small short-lived memory objects, which is in contrast to imperative languages, where most data is mutable and memory areas more often may be larger or live longer. This has an impact of which memory usage scenario the memory management system should be optimized for and thus may influence what makes a good memory management strategy. This may be taken into account, when choosing a strategy. However, if the strategy is good enough to handle all possible scenarios, this would be even better. This was formulated as goal number 10 in section 1.3

- **Amount of available type information**: This project emphasizes to some extent on typed memory management, which mostly makes sense for strongly typed programming languages, such as Standard ML, Java, C# and other .NET languages. For example, weakly typed languages, like C, has untyped pointers (type `void *` in C). Also, it is significant whether the language is statically (as e.g. Standard ML) or dynamically (as e.g. Python) typed, since for dynamically typed languages, the specific types are not available until runtime, which means that many opportunities for compile-time optimizations are lost. Finally, the types are only available to the memory manager, if the compiler preserves them during its compilation. It is possible to preserve types for strongly typed languages, but for weakly typed languages, the types may never have existed, which could make it seriously problematic. It is fairly easy to preserve types for statically typed languages, but for dynamically typed languages, it will require generation of code for runtime passing of types or runtime casing-out on types. These considerations are especially important if the memory management strategy *relies* on using types for achieving interactivity or high performance. Hence, certain of the memory management strategies investigated in this thesis are likely to be less suitable for languages or compilers with less type information available. A particularly relevant case is compilers which handle polymorphic types throughout the compilation and at runtime, since this makes it more difficult and likely less efficient to acquire type information at all points in a program. This has specifically been avoided in the compiler used in this project. On the other hand, type information is mostly only used for optimizations in this thesis, meaning that types are not necessarily relied heavily upon

- **Experimental bias due to choice of language:** The choice of language makes the experiments in this thesis biased towards functional languages. For example, according to three articles mentioned on page 46 in section 3.2 in [Jone96], the pointer stores on modern optimizing compilers for languages like Lisp and ML may be as rare as 1% of the instructions and as will be described next in section 4.1, mutable data and update operations are challenging for garbage collection methods

Chapter 4

Garbage Collection

This section goes through garbage collection methods in general. In particular, it analyses the basic garbage collection methods in detail and goes through some extensions.

4.1 Mutable Data

Mutable data presents a challenge to garbage collection methods. The following can be said:

- **Older allocated memory pointing to newer allocated memory**: Mutable data may cause older allocated memory to point to newer allocated memory. This can only happen if some memory location is actually *updated*, after it and some other memory has been allocated. Hence, this problem can only exist for functional languages whenever updating references or other mutable data structures, such as arrays. For some functional languages with lazy evaluation, it may also happen for suspended calculations (thunks) and for some implementations, it may also happen in other places in the compiler. However, the two latter examples here are actually not applicable to the language considered for this thesis

- **Program behaviour for lists**: In functional languages, lists normally have their elements added to the beginning of the list. In imperative languages, it is common to update the last null-pointer of the list, when adding elements, for keeping the list elements in the order they were added. This typical imperative approach makes older allocated memory point to newer allocated memory

- **Not "garbage collection friendly"**: Mutable data is not particularly garbage collection friendly. For example, if the mutator updates values while the collector performs garbage collection (e.g. in a concurrent collector), the update may invalidate part of the already processed (by the collector) part of the heap. The descriptions above mention older allocated memory pointing to new allocated memory. For collectors with special treatment of more than one generation (in age) of memory objects, the back-reference from older to newer generations is normally costly and requires extra processing. Also, only via references from older to newer memory objects is it possible to create cycles, which is problematic for the specific method of reference counting. For some compilers of functional languages, cyclic data structures may also be created by closures for recursive functions, but this is not the case in the implemented compiler

4.1.1 Roots and Pointer Finding

For performing garbage collection, it is necessary to trace the entire heap, which starts at its root set, which consists of all variables on the stack and in registers with pointers to the heap. Tracing the root set typically requires a tight coupling between either the compiler and the memory-management system or the mutator and the memory-management system:

- **Compiler cooperation**: Tracing collectors, particularly copying collectors, need cooperation from the compiler. Mark-sweep collectors need less cooperation, but still require it (see section 2.5 page 31 in [Jone96]). It is claimed in section 2.5 page 38 in [Jone96] that it is possible to do reference counting without any compiler support. However, section 2.5 page 38 and section 3.5 page 67 in [Jone96] refer to Christopher's reference counting library [Chri84] as an example of this, but this is for the language Fortran and is implemented as a library, not part of a compiler, which means that it likely modifies or overloads certain calls or similar, which, in effect, is a change of the implementation of certain basis calls and operations of the language. Hence, doing reference counting is not in general possible without compiler support. In general, for all memory management methods, the more advanced they become, the more cooperation they typically require from the compiler.

  This consideration would be relevant, if we were trying to make a memory management system, which would work without or independently of a compiler. However, this is not the case. In this thesis, we even intend to take advantage of static type information, which surely will require compiler support. Hence, we do not regard the requirement for compiler cooperation as a limitation. This was stated earlier as goal number 8 in section 1.3

- **Coupling to mutator**: The reference counting method is tightly coupled to the mutator. This gives two primary costs: 1) pointer operations are expensive and 2) reference count invariants must be preserved accross program changes (section 2.5 page 38 in [Jone96]).

  This consideration would be relevant, if, for example, we had been compiling program modules independently, with shared memory usage between the modules. However, the memory management system being designed is intended for a whole-program optimizing compiler, where separate compilation of modules is not an issue. The consideration would also be relevant, if one is to compile separate DLLs (dynamic link libraries) or similar compiled modules, whose memory usage interacts with the program (or another DLL) which uses the DLLs in question. Hence, this consideration is important, at least with respect to support for compiling into DLLs (goal number 9 in section 1.3)

The actual pointer-finding can be done in at least two different ways:

- **Exact pointer-finding**: This is the most common way and is the only method considered in this thesis; it consists of tracing all register, stack and global variables which are pointers and only pointers

- **Conservative pointer-finding**: This is a method typically used for automatic memory-management methods for languages like C, where not all variables have well-defined types, since e.g. C has pointers of types `void *`. The method normally treats all variables, which could potentially be a pointer, as if it was a pointer. Thus, integers may occasionally be treated as pointers. This method is not considered in this thesis; on the contrary, we seek to take advantage of type-information and expect at least pointers to be statically classified

4.2 The Tricolour or Tetracolour Abstraction

The tricolour abstraction is due to Dijkstra et. al. [Dijk78] and is often used for representing the state of each memory object or area in a heap. A memory cell's colour, black, grey or white, changes throughout execution of a program and with garbage collection. In some settings, particularly for Baker's Theadmill, presented later in section 6.6, a fourth colour is also used: off-white. The colours are shown in figure 4.1 and their meanings are summarized here: [1]

1. **Black**: The memory cell and its immediate descendants have been visited by the garbage collector. The garbage collector has finished with black cells and need not visit them again
2. **Grey**: The garbage collector must visit grey memory cells again. Either the memory cell has been visited but its components may not have been scanned or, in an incremental or concurrent setting, they may have been subject to modification by the mutator in a way that has rearranged the connectivity of the memory object graph
3. **White**: These memory cells are unvisited and, when garbage collection is complete, are garbage
4. **Off-white**: These memory cells are garbage and have been converted to available free memory. This is usually referred to as a free-list

Figure 4.1: The way that the four colours, black, grey, white and off-white, of the tricolour or tetracolour abstraction will be shown in this thesis

Garbage collection is generally considered complete when all reachable nodes have been scanned (blackened), i.e. there are no unscanned grey nodes left. All white nodes at this point are garbage and may be reclaimed. The white nodes can be made off-white at this point, but most methods refer to the concept of free-lists or just free memory, rather than the colour off-white.

4.3 Space Overhead of Dynamic Garbage Collection Methods

Dynamic garbage collection methods generally have space overhead. A short and general overview of space overhead for the widely known methods is the following:

1. All widely known methods (mark-sweep, copy collection and reference counting): The garbage collection is recursive in nature, which requires stack-space for performing the actual collection (section 2.5 page 38-39 in [Jone96]).

   Possible *solutions* and *improvements*:

   (a) Cleverness in the algorithms is the typical solution to this, where the specific kinds of cleverness depend on the method. Many of the common methods can overcome this disadvantage:

[1] Lin's partial mark-sweep algorithm for reference counting (presented in section 3.5 p. 62-65 in [Jone96]) additionally uses the colour purple, but this is specific to handling cyclic data structures in that method, so it is not very useful as a general abstraction

Copy collection can use Cheney's method of forwarding-pointers in Fromspace, mark-sweep can use pointer reversal (although this requires extra permanent space for all memory objects with child-pointers, see section 4.3 pages 82-87 in [Jone96] or algorithm 3.6 in section 3.1 in [Appe98]) and the recursive freeing in reference counting can be done with tail-recursion, thus not requiring stack space

(b) Static analyses of types may determine specific memory layouts for certain data structures, such that specialized functions can be used instead of recursion. This can only be considered as an improvement, not a solution, since dynamically recursive data structures normally exist and must be supported. Also note that such specialized functions take up space, as we will see later

2. All methods: Overhead in tracking nested pointers in data structures. Typically, a tag, a header word or similar is used to distinguish pointers from values, in order to be able to traverse memory.

   Possible *improvements*:

   (a) By using static type information, the tags can sometimes be avoided entirely for distinguishing pointers from data. However, this requires specialized garbage collection functions to be generated and that these be attached e.g. to the stack frames, thus likely causing another memory overhead instead. It seems advantageous though to have a memory overhead proportional to the stack size, rather than proportional to the heap size. Also, specialized garbage collection functions avoid runtime testing of memory tags, thus likely giving a slight performance gain as well

   (b) The heap could also be segregated according to the kinds of memory objects, where the need for the distinction between pointers and values could be avoided entirely for e.g. memory objects being purely values without pointers. Such objects are easily identified by static types. One disadvantage of this is poorer locality of reference, degrading cache behaviour

3. Mark-sweep and copy collectors: These methods require extra head room, in order to be efficient, since collection would otherwise occur too frequently (see later in sections 4.5 and 4.6)

4. Reference counting: The reference counters give overhead

5. Mark-sweep collectors: The tables or flags or bitmaps for marks give overhead

6. Copy collectors: Twice the size of the maximum amount of resident memory is required

The last three kinds of space overhead seem to inherently be part of the methods and there does not really seem to be any solutions to this, except possibly the last one. When considering more advanced methods, such as in particular incremental garbage collection, there will generally be more overhead, but this will depend more specifically on the method.

Performance improvements often give space overhead, so it is hard to quantify or analyze from a general point of view. Space overhead is also the least important in this thesis (see goal number 6 in section 1.3), so in this thesis, more focus will be on the execution-time performance of methods, rather than their space overhead.

4.4 The Reference Counting Method

The reference counting method was briefly introduced in section 1.2.1 as an explicit memory management method keeping track of memory areas by maintaining a counter of the number of references to each

allocated memory area. This section gives more information about the properties of the method, mostly in terms of its advantages, disadvantages and some of the methods for improving it.

4.4.1 Advantages of Reference Counting

Reference counting has the following advantages:

1. **Good cache behaviour**: Reference counting has good cache behaviour (locality of reference) according to section 2.1 page 23 in [Jone96]. However, over time, as an application is running, this advantage diminishes with memory fragmentation, if nothing is done about that

2. **Only one copy of the memory**: It only requires one copy of the memory contents, whereas e.g. copying garbage collectors require being able to store two copies of the used memory

3. **Immediacy, tighter data life-time management**: This is either *desirable* or *required* by most applications (section 2.5 page 36 in [Jone96]). It can also give "better" support for finalization, since e.g. scarce resources may be freed earlier (section 2.5 page 36-37 in [Jone96])

4. **No performance degradation with the amount of live data**: The other common dynamic memory management methods (mark-sweep and copying collection) generally degrade in performance proportially to the amount of live data, whereas reference counting does not (section 2.5 page 39 in [Jone96]). The only possible exception to this is an indirect degradation, due to memory fragmentation (listed as a separate disadvantage), e.g. if the program has been running for a long period of time

5. **Can naturally be made incremental**: Most other garbage collection methods require an additional effort to become incremental, or get additional disadvantages in becoming so. Reference counting seems more natually suited to becoming incremental, although it does get additional overhead to the already expensive overhead of updating pointers. This overhead is constant though and the algorithm therefore remains incremental in nature

6. **Does not need to move memory objects**: Copying memory objects around takes time, especially for large objects. Reference counting does not normally need to move objects around. However, for avoiding fragmentation, it may be necessary to do so, at least occasionally

4.4.2 Disadvantages with Possible Solutions and Improvements

There are, however, also some disadvantages of reference counting. Some disadvantages have known solutions or improvements, but notice that, some of the solutions or improvements may either be mutually exclusive or lose some of the advantages of the method:

1. **High overall process cost**, (section 2.5 page 38 in [Jone96]): Some of the costs occur on the following occasions:

   (a) *Moving a pointer*: It normally only takes one register move operation to move a pointer, but four memory accesses are needed, in order to increment and decrement the reference counters

   (b) *Non-destructive pointer traversal*: Even non-destructive operations like traversing a list may require reference counters to be incremented and decremented (p. 45 section 3.2 in [Jone96])

Possible *improvements*:

(a) *Deferred Reference Counting*: With this method, no references are counted from local variables or stack-allocated compiler temporaries; only references from heap objects are counted. Deferred Reference Counting requires some means of keeping track of when to free objects whose reference counter is zero. Methods for this are *zero count tables* (ZCT) (section 3.2 p. 46 and algorithm 3.3 p. 47 in [Jone96]) or the temporary increment approach, buffering and coalescing from [Blac03b]

Deferred Reference Counting however *reduces the advantage of immediacy* (p. 50 sec. 3.2 in [Jone96]), but may increase its performance by 80% (with ZCT) at the cost of a small space overhead, e.g. 25kb (p. 49 sec. 3.2 in [Jone96])

2. **Occasionally causes collection pauses**: The following occasions are the sources of pauses:

(a) *Freeing complex data structures*: When a reference counter reaches zero for a memory object, which refers to many other memory areas, usually by a whole graph of references, the entire data structure may need to be freed recursively. This may take significant amounts of time, if done eagerly (section 2.5 page 36 in [Jone96]).

Possible *solutions*:

i. A lazy freeing algorithm may be used, where only the immediate object is freed and put onto a free-stack, using the now zero reference counter field to store the stack list pointer. When a freed object is allocated again, its immediate descendants are freed. However, *the advantage of immediacy is lost* (see section 3.1 page 44 in [Jone96])

(b) *Free list searching*: Allocation with reference counting involves managing free-lists. These can cause pauses when searched for a free memory block.

Possible *improvements*:

i. *Improved free list data structures*: May reduce pause times, see later in section 4.4.3

ii. *Occasional compacting of memory areas*: This might be possible and at the same time solve the fragmentation problem, but I don't have any references for this. This may, however, come at the cost of losing some of the advantages of the method, particularly that memory objects do not need to move

3. **Memory fragmentation**: Deallocation and allocation causes memory fragmentation (section 2.5 page 38 in [Jone96]). The *improvements for free list searching* above should help here as well

4. **Cyclic data structures are not freed**: This is a well-known property of reference counting.

Possible *solutions* and *improvements*:

(a) It can be handled by combining reference counting with another kind of garbage collector, e.g. a mark-sweep collector, which when combined with *sticky reference counting* may even reduce the space overhead for the reference counters. These methods are explained in section 3.3 on pages 50-51 in [Jone96]

(b) Cycles can be checked for explicitly, which, however, is likely to give space or time overhead, possibly even pauses during collection, unless done by a time-bounded method like *trial deletion*, mentioned in section 1 of [Baco03], with reference to [Baco01]

(c) In functional languages, cycles can only occur at certain places, e.g. only for reference and array types for the compiler considered in this project, so handling and checking for cycles can be restricted to such cases

(d) The following methods are described in section 3.5 pages 56-68 in [Jone96], but it is also stated that none of these methods have been adopted for use by any significant systems: Bobrow's technique, weak-pointer algorithms (which historically have had problems, whose corrections come at a considerable cost) and partial mark-sweep methods

5. **Space overhead for reference counters**: This is a general overhead of the method, but as argued previously in section 4.3, other methods have a similar overhead, one way or another, so this seems like an inevitable disadvantage.

   Possible *improvements*:

   (a) *Sticky reference counts*: This method uses only a one-byte reference counter. Reference counting is stopped when it reaches 255: the sticky value. A tracing collector can occasionally collect the sticky referenced objects, as well as cycles (section 3.3 pages 50-51 in [Jone96])

   (b) *One-bit reference counts*: Tagged pointers may be used to track references to non-shared memory objects (and few memory cells are shared, as argued previously in section 3.3), but this method has to be combined with other methods, e.g. tracing mark-sweep collection, to give a comprehensive memory management system (section 3.3 pages 51-53 in [Jone96])

4.4.3 Free Lists and Free-Space Management

Several memory management methods, e.g. reference counting and mark-sweep, need to be able to manage free lists. Even manual memory management requiring programmers to use explicit allocation- and free-functions have to do this. If they are managed simply as lists of free memory areas, problems like fragmentation and long allocation times likely occur. Some ways of improving on this could be:

- **Size class segregated free lists**: Free lists can be organized into multiple lists of differing sizes. The size classes could be in powers of two or in increments of a factor of $1+\frac{1}{8}$, as used in [Baco03], or a specially designed set of size classes to bound the internal fragmentation to at most $\frac{1}{8}$ (i.e. 12,5%), as in [Blac03b]. With such a segregation into size classes, the appropriate list can be searched when requesting space for a memory object of a given size. If one list does not contain any free memory blocks, lists for larger sizes can be searched, where the memory block found can be split into smaller blocks and remaining (unallocated) blocks put to the free lists for the appropriate sizes. Segregating free lists according to size, however, sacrifices locality of reference, since memory objects cannot be ordered in memory according to their creation order, as will be argued later in section 4.9 to be generally advantageous

- **The buddy system**: Free lists can be organized as a binary search tree, such that blocks become sorted according to their memory addresses. This hierarchically divides the blocks according to size and gives both faster search times and an efficient way of merging neighbouring memory blocks ("buddies"), when the tree is traversed. This system is described in detail in [Knut97] and in chapter 12 in an upcoming version of [Moge09]

4.5 The Mark-Sweep Method and its Relatives

The mark-sweep method was briefly introduced in section 1.2.3 as an implicit tracing memory management method with two main phases: 1) mark all memory cells reachable from all roots and 2) sweep the entire heap, freeing all unmarked cells. This section gives more information about the properties of the method, mostly in terms of its advantages, disadvantages and some methods for improving it.

4.5.1 Advantages of Mark-Sweep and its Relatives

1. **Good cache behaviour**: Mark-sweep has some of the good good cache behaviour (locality of reference) of reference counting, particularly when using mark-compact methods (chapter 5 in [Jone96]), since e.g. often, the memory layout of the allocation order is preserved (recall from section 3.3 (point 3) why this is important)

2. **Only one copy of the memory**: It only requires one copy of the memory contents, whereas e.g. copying garbage collectors require being able to store two copies of the used memory

3. **Low mutator processing cost**: There is no cost when the mutator updates pointers or performs other activities, so time is only spent on the actual garbage collection. This is in contrast to reference counting (p. 26 section 2.2 in [Jone96])

4. **Cycles are handled natually**: The mark-sweep method is comprehensive, thus not leaving any floating garbage (p. 27 section 2.2 in [Jone96])

5. **Does not need to move memory objects**: As was the case for reference counting, mark-sweep does not normally need to move objects around. For avoiding fragmentation, it may however be necessary to do so, at least occasionally

4.5.2 Disadvantages with Possible Solutions and Improvements

1. **Not incremental in nature**: Marking and sweeping the entire heap at once is not an incremental approach and therefore leads to garbage collection pauses (p. 27-28 section 2.2 in [Jone96])

   Possible *improvements*:

   (a) It is somewhat easy to incrementalize at least the sweep phase: the mark bits and memory objects being put into free lists are invisible to the mutator (p. 89 sec. 4.5 in [Jone96]). The three methods Hughes' lazy sweep algorithm, the Boehm-Demers-Weiser sweeper and Zorn's lazy sweeper from pages 89-93 in [Jone96] all do an amount of sweeping during allocation and achieve free list management at the same time

2. **Collection time-complexity is proportional to the heap size**: The sweep phase traverses all memory cells, which takes time proportional to the heap size (not just the size of the resident memory) (p. 28 section 2.2 in [Jone96])

   Possible *improvements*:

   (a) When using more advanced methods, the asymptotic time-complexity may be no greater than that of copy collection (p. 88 sec. 4.5 in [Jone96] and p. 93-94 sec. 4.6 in [Jone96]). E.g. marking and adding memory objects to a free list is cheaper than copying them, particularly for large objects. Also, when using mark bitmaps, the heap does not even need to be touched

3. **Memory fragmentation**: (p. 28 section 2.2 in [Jone96]).

   Possible *solutions* and *improvements*:

   (a) The Mark-Compact method(s?) is(are?) well-known for solving this... When compacting memory by mark-compact methods, this is likely to even become an *advantage* over copy collection, since compacting memory in this way is likely to preserve the original allocation order of memory (see page 137 section 6.7 in [Jone96]). Also see the section 4.9 regarding memory layouts and the order of traversing and copying memory. Another advantage of mark-compact is that, it can make *allocation as efficient as for copy-collection*, which is basically as fast as it gets

   (b) Without mark-compact, the improvements for free-list searching apply here as well as for the reference counting method, see section 4.4.3

4. **Requires head room to be efficient (performance degrades with higher recidency)**: Higher residency (occupancy) requires more frequent garbage collection, which is why extra head room is needed for avoiding this (p. 28 section 2.2 in [Jone96])

4.6 The Copy Collection Method

The copy collection method was introduced in section 1.2.2 as an implicit tracing memory management method, which divides the heap into two semi-spaces: Tospace, which contains the current active memory, and Fromspace, containing obsolete data. Garbage is collected by flipping the roles of the two spaces and copying only the live data from one to the other. This section gives more information about the properties of the method, mostly in terms of its advantages, disadvantages and some of the methods for improving it.

4.6.1 Advantages of Copy Collection

1. **Low mutator processing cost**: There is no cost when the mutator updates pointers or performs other activities, so time is only spent on the actual garbage collection. This is in contrast to reference counting (p. 29 section 2.3 in [Jone96])

2. **Cycles are handled natually**: Like the mark-sweep method, copy collection is comprehensive, thus not leaving any floating garbage

3. **No memory fragmentation**: The copying of only live data naturally compacts the memory, thus avoiding fragmentation (pages 29 and 31-32, section 2.3 in [Jone96])

4. **Efficient memory allocation**: It is hard to see how allocation could be done more cheaply, since it only involves incrementing a pointer and comparing it with another pointer, where the comparison is to determe whether enough memory is available (pages 29 and 31-32, section 2.3 in [Jone96])

4.6.2 Disadvantages with Possible Solutions and Improvements

1. **Not incremental in nature**: Copying the entire heap at once is not an incremental approach and therefore leads to garbage collection pauses, as for mark-sweep

2. **Needs to move memory objects**: The usual copying algorithms move all live memory around between each semi-space flip

   Possible *improvements*:

   (a) Non-moving or mostly-non-moving garbage collectors improve on this. These methods are a divergence from the basic copy-collection method

3. **Collection time-complexity is proportional to the amount of live data**: The copying traverses all live memory cells, which takes time proportional to the amount of live memory. This is better than mark-sweep, but still worse than reference counting

   Possible *improvements*:

   (a) Larger objects take a significant time to copy. If these are handled specially, e.g. by a small header and a pointer to a larger memory area, managed by a non-moving memory manager, much of this copying overhead can be avoided. If sufficient operating system support is available, large objects can also be assigned their own virtual memory pages, where they can have their addresses re-mapped instead of being copied (p. 138 in section 6.7 in [Jone96])

4. **Bad cache and paging behaviour**: The cache behaviour (locality of reference) is bad, due to the copying of data, but even worse, the paging behaviour is also bad, due to the copying between the two semi-spaces (p. 32 section 2.3 in [Jone96] and p. 93 section 4.6 in [Jone96])

   Possible *improvements*:

   (a) The order of copying the memory objects is significant and may give some improvements, as will be argued later in section 4.9. However, it does not solve the inherent disadvantages of the copy collection method as just described

5. **Requires head room to be efficient (performance degrades with higher recidency)**: As for mark-sweep, higher residency (occupancy) requires more frequent garbage collection, which is why extra head room is needed for avoiding this. When also considering the next disadvantage below, performance likely starts to degrade earlier than for mark-sweep in programs which use up the entire system's heap memory, since when twice as much memory is needed, the residency (occupancy) will start to be stressed earlier (something similar but less accurate is stated on page 33 in section 2.4 in [Jone96])

6. **Requires twice the amount (two semi-spaces) of allocated memory (and head room)**: This memory cost is quite high, compared to the other methods (p. 28 section 2.3 in [Jone96])

4.7 Common Properties of Tracing Garbage Collectors

Tracing garbage collectors, such as mark-sweep and copying collectors, share certain properties, some of which are listed here:

1. Roots and pointer finding normally must be determined precisely, as described earlier in section 4.1.1, in order to be able to trace pointers

   Possible *improvements*:

   (a) Chapter 9 in [Jone96] relaxes this need, as do non-moving tracing collectors

4.8 Differing Properties of Tracing Garbage Collectors and Mark-Region

The three canonical methods of tracing garbage collectors are historically considered to be: mark-sweep, mark-compact and semi-space copy collection. According to [Blac08], each of these sacrifice one performance goal, specified as follows:

1. Mark-sweep has poor locality and slow mutator time, due to its non-contiguous allocation. This mostly comes from heap fragmentation, when the program has been running for a longer time. This would likely be the same for reference counting, although they do not state that
2. Mark-compact has slow collection time, due to its compacting
3. Semi-space copy collection is space inefficient

The article [Blac08] state these sacrifices in relation to their method, *mark-region*, which supposedly meets all three performance goals: mutator performance (due to the better locality), space efficiency and fast collection. Notice however that, this does not mention pause times, which is important in this thesis. Their mark-region method, as well as their instance of it, Immix, will be presented further in section 6.9

4.9 Heap Layout and Traversal Order for Various Methods

The way that various methods organize and traverse objects in memory has an impact on the performance with respect to cache and paging behaviour. The following lists some possible memory layout strategies:

1. **Breadth-first**: The commonly used semi-space copy collector, Cheney's algorithm, which was presented in section 1.2.2, normally uses breadth-first traversal. This organizes the memory objects in the heap in a breadth-first manner after each copy collection
2. **Depth-first and approximately depth-first**: Cheney's copy collection algorithm may be altered to use Chase pointer reversal, in order to achieve semi-depth-first copying (algorithm 3.11 in [Appe98]). An alternative is to use Moon's approximately depth-first copying (as shown in algorithm 6.4 on page 139 in [Jone96]). There also exist other depth-first methods. According to page 135 in section 6.6 in [Jone96], using approximately depth-first has been seen to increase garbage collection elapsed times by 6%, but there is no report on how it affected page faults (this is in comparison to breadth-first, I believe). According to references on pages 132-133 in section 6.6 in [Jone96], depth-first only performs slightly better than breadth-first. According to a reference on page 135 in section 6.6 in [Jone96], page faults were reduced by 15% by recursive depth-first copying, in comparison to breadth-first. It should be noted that several of the depth-first methods have overhead in either requiring tag-bits or a recursion stack, which gives some overhead in itself, so it may be hard to compare these methods with all aspects taken into account
3. **Hierarchical decomposition**: One way of copying objects is to copy them in blocks of objects with their immediate children, which decomposes the memory data structure into groups of smaller related components. On page 136 in section 6.6 in [Jone96], it is claimed that this is often better than depth-first copying and approximately depth-first. However, all these observations depend on the data structures held in memory and how they are used by the application (the mutator) (pages 136-137 in section 6.6 in [Jone96])

4. **Special casing on certain data structures**: Since many of the above observations depend on the data structures used by the mutator, it may be advantageous to handle certain well-known data structures specially. Some of the references on pages 132-133 in section 6.6 in [Jone96] state that, treating hash table data structures as a special case has been advantageous for Lisp programs. Such special casing will generally not be considered or encouraged in this thesis, as it violates the intentions of being able to use the same memory manager for different types of programs or programming languages (points 10 and 11 in section 1.3)

5. **Maintaining creation order**: If the garbage collector can maintain the order that the objects were allocated in, when organizing its heap layout, this gives fewer page faults than breadth-first and depth-first, both of which had fewer page faults than random order (according to page 132 in [Jone96]). The creation order of memory objects seems to be the only "universally beneficial" order of memory layouts. Sliding compactors from chapter 5 in [Jone96] preserve the heap layout, but copy collectors do not. The mark-region method from [Blac08], which will be presented in detail in section 6.9, has as one of its advantages that it maintains the creation order of objects

6. **Static or dynamic regrouping**: The garbage collector may actively regroup data, as described on pages 131-132 in section 6.6 in [Jone96]. This, in combination with adaptive training (as described on page 208 in section 8.5 in [Jone96]), has been seen to reduce paging times by 65-80%

4.10 Hybrid and Generational Garbage Collection Methods

As we have seen in the previous sections, the basic garbage collection methods all have their advantages and disadvantages. Some disadvantages can be improved upon or even removed, but usually not all at the same time. Also, some improvements remove or reduce some of the advantages or have other costs, such as e.g. extra space overhead. For these reasons, it is relevant to consider combining methods, thereby forming *hybrid methods*. This section introduces hybrid methods and we shall see several examples of such methods in the sections that follow.

4.10.1 Generational Garbage Collection

One way to form hybrid methods is to divide the heap into *generations* of objects, according to the age of memory objects. However, dividing the heap into generations does not necessarily imply forming hybrid methods. The heap can also simply be divided into two or more generations, all managed by the same garbage collection method.

It is often important and very beneficial to divide the heap into at least two generations. In such cases, the part of the heap containing the youngest objects is typically known as the young generation or the *nursery* generation. The generation of older memory objects is often referred to as *mature-space*.

Key to implementing generational garbage collection is the need to implement *memory-barriers*, which is the main topic of section 4.11. That section also introduces *remembered sets*, which are also typically used for generational garbage collection. Remembered sets and in particular memory-barriers also have other applications than forming generational methods, such as making methods incremental.

Generational collection is mostly an improvement method, if the same method is used for all generations. It can reduce the frequency of performing full-heap collections, where the partial collections are usually faster, thus also reducing the frequency of long pauses, but not the worst-case pauses. The reason why this works well in practice is due to the generational hypoteses, as mentioned in points 1

and 2 in section 3.3. The weak generational hypothesis states that many objects are short-lived and the strong generational hypothesis states that long-lived objects die or are modified rarely.

4.10.2 Ways of Forming Hybrid Methods

The following is a list of some ways in which collection strategies may be combined:

1. **Heap generations**: As described in the previous section, the heap may be segregated into multiple heaps according to the age of memory cells residing in the areas. If different generations use different garbage-collection *methods*, hybrid-generational garbage collectors arise. One of the very common ways of doing this is by handling a young (nursery) generation and a mature generation of older objects by different methods. Splitting into such two generations gives the method used for the nursery generation the opportunity to be optimized for the assumption that many of its objects are short-lived, while the mature generation can be made suitable for long-lived objects. Many of the presented methods split into such two generations and are sometimes referred to as two-generation hybrid methods

2. **Separate handling of large objects**: Large objects pose several problems, as explained in e.g. section 3.5 in [Baco03]. One problem is that there is a large overhead for copying them, which takes an unbounded amount of time, which is problematic in interactive or real-time settings. Another problem is that they may be impossible to allocate in a fragmented heap. For example, a single long-lived object in the middle of the heap or the middle of an allocation area reduces the maximum available contiguous memory to half the size. Section 6.3 p. 126 in [Jone96] contains references suggesting to handle large objects as a header with a pointer to the actual large object, which can then be managed by a non-moving garbage collector, e.g. mark-sweep, in order to avoid copying the object, although occasional compacting may be necessary. Most of the presented methods handle large objects separately somehow

3. **Separate static area heap**: Objects which are global or immortal, possibly part of the memory manager itself, can be stored in a separate heap. This may include global objects, which are either immutable or which are mutable but remain constant in memory size and layout. For a language like ML and a type-preserving compiler, such objects may be easily classified as e.g. having either no references or arrays, or having references (or arrays) containing no datatypes with any more than one constructor. This method is generally referred to as having a *static area* in section 6.3 p. 126 in [Jone96]. Global or immortal data are long-lived and since the memory residency and the amount of live objects degrade the performance of many methods, this optimization is highly recommended. Some methods, like Ulterior Reference Counting presented in section 6.10.1, even *requires* a separate heap for immortal objects. It may however in some cases be a problem to determine which objects are global or static or immortal

4. **Handling mutable objects separately from immutable objects**: In functional languages like Standard ML, mutable objects include only references and arrays and it is known at compile-time, which values may contain such objects. This gives the opportunity to place such objects in a separate allocation area and handle them specially, which makes a kind of hybrid method. The reason for doing this is that mutable objects usually pose challenges to garbage collection, as mentioned in section 4.1 and as will become even more clear in the next sections, when memory-barriers will be discussed

5. **Handling objects of certain static types separately**: Certain properties about memory objects may be known at compile-time, such as whether they have child-pointers or not. The point above mentioned mutable versus non-mutable objects. In general, static analysis or types may classify memory objects at compile-time. This may be used to form hybrid methods, by handling certain types of objects by different methods

The above list is a set of some of the generally useful methods and tricks, which are used by several of the algorithms presented. It is actually surprising that the book [Jone96] does not introduce hybrid methods more formally than occasional mentioning of it, e.g. on its page 138 in its section 6.7. Many high-performance garbage-collection methods with short pause times are hybrids of some form, as evidenced by the methods presented in the following sections.

4.11 Remembered Sets and Memory Barriers

Some of the general and commonly used methods when forming more complex garbage collectors are remembered sets and memory barriers. Memory barriers protect parts of the heap against either read or write operations and perform some specific task when such reads or writes occur. A simple and common task to perform for a write-barrier is to maintain a remembered set, i.e. a set of objects known to have written to the heap. Barriers may also perform other operations, but remembering a set of memory objects is common. There are various ways of representing remembered sets, such as by a linked list, but they all achieve the same simple task: to remember a set of objects. Memory barriers and the reasons for having them are explained further in the following sections.

4.11.1 Memory Barriers

Memory barriers generally come in two flavours, read-barriers and write-barriers. They are normally used for tracking the mutator's disruption of garbage collection:

- **Read-barriers**: Ensure that the mutator never sees white objects. The read-barrier action is triggered when white objects are accessed
- **Write-barriers**: Record where or perform an action whenever the mutator writes black-to-white pointers, to ensure that the collector (re)visits the relevant memory nodes

The barriers try to prevent falsely reclaiming live objects as garbage. This can only happen if a white object becomes invisible to the collector, while still being reachable by the mutator. This means that *both* of the following two conditions must hold (section 8.2 page 187 in [Jone96]):

- **Condition 1**: A pointer to a white object is written into a black object and
- **Condition 2**: The original reference to the white object is destroyed (i.e. the reference in condition 1 is the *only* reference)

Write-barrier methods normally work by making sure that one of these two conditions for failure cannot happen. [Wils92] classifies write-barrier methods as either *snapshot-at-the-beginning*, which prevents condition 2 from occurring, or *incremental-update*, which records when condition 1 happens. Write-barriers are much more common than read-barriers and both kinds of barriers are described in detail in the following sections.

4.11.2 Read-Barriers

Read-barriers are typically implemented in software by the compiler emitting a few instructions at pointer reads. Three examples of specific read-barriers are 1) masking out bits of pointers (unconditional), 2) the Brooks-style read-barrier (unconditional) used in e.g. the Metronome system from [Baco03] and 3) the evacuating read-barrier (conditional) from Baker's algorithm, used for forming an incremental version of Cheney's copy collector. Baker's algorithm will be briefly reviewed later in section 4.12.1 and the Metronome system will be reviewed further in section 6.7.

General read-barriers will be presented in a way similar to [Blac04b], where an unconditional bit-masking barrier and a conditional barrier are shown. In [Blac04b], the barriers are shown as Java code, as PowerPC assembly instructions and as 32-bit x86 assembly (IA-32) instructions. In the presentation here, the source code will instead be shown as Standard ML code with only the corresponding x86 assembly instructions shown. The reader is referred to [Blac04b] for the Java and PowerPC versions.

The skeleton source code in Standard ML for presenting the read-barriers is shown in figure 4.2. This is shown as a function named `readBarrier`, which *should* be inlined, as also stated in section 4 in [Blac04b]. The article [Blac04b] generally refers to the inlined part of a barrier as the *fast path* and any non-inlined part as the *slow path*, which goes through a function call. The importance of this distinction is stressed in [Blac04b]. The code in figure 4.2 also uses a type `addr`, which is assumed to be an opaque `word` type of the same width (32-bits or 64-bits) as an address. The functions `UnsafeMemoryAccess.addrtoword` and `UnsafeMemoryAccess.wordtoaddr` convert between opaque `addr` values and non-opaque `word` values, assumed to have a corresponding `WordAddr` module; the compiler should normally be able to optimize away these two calls. The `WordAddr` is either the same module as `Word32` or `Word64`, depending on the architecture, but it should be assigned to a structure named `WordAddr`, in order to somewhat increase the portability of code operating on the low-order bits of pointers. A loose specification of the module *UnsafeMemoryAccess* can be found later in section 8.3, particularly in the figure 8.4.

```
(* The readBarrier function should be inlined *)
fun readBarrier(obj : addr, slot : addr, mode : int) : addr =
      let
        val opaque = UnsafeMemoryAccess.readaddr(slot, 0)
        val value  = UnsafeMemoryAccess.addrtoword(opaque)

        val modifiedValue = value (* Replace this line with the read-barrier code *)
      in
        UnsafeMemoryAccess.wordtoaddr(modifiedValue)
      end
```

Figure 4.2: Read-barrier skeleton code in Standard ML. One of the individual barriers in figure 4.3 are embedded into this by replacing the line with the comment. A loose specification of the module *UnsafeMemoryAccess* can be found later in section 8.3, particularly in the figure 8.4. The compiler should be able to optimize away the calls to the functions *addrtoword* and *addrtoword* in this module

The code for the two kinds of actual read-barrier implementations are given in figure 4.3, where both the Standard ML code and the 32-bit x86 assembly (IA-32) instructions code are shown. The code from this figure is to be inserted in the skeleton code in figure 4.2, where it replaces the line with the comment. The x86 assembly presented here uses the GAS (GNU Assembler) syntax, which specifies source operand first, then destination, which is opposite of the presentation in [Blac04b]. The constant

$-4 in the generated assembly code corresponds to the bit-wise negation of 0wx03. This is an equally portable way of specifying that constant with respect to 32-bit versus 64-bit pointers. It is, however, generally less important for assembly code to be portable, most certainly if it is generated automatically from portable source code, such as the Standard ML code shown.

The conditional read-barrier shown in figure 4.3 does not perform any computation. The Metronome system from [Baco03] uses a Brooks-style read-barrier, which would look differently and thus generate different code. The point of the article [Blac04b] and the figures shown here are, however, to illustrate the performance and general overhead of barriers, not to implement all possible specific barriers. Important specific barriers will be shown next.

Standard ML	**GAS x86 Assembly (IA-32)**
Unconditional read-barrier	
`val modifiedValue =` `  WordAddr.andb(value, WordAddr.notb(0wx3))`	`and  $-4 %eax`
Conditional read-barrier	

`val modifiedValue =`
`  if WordAddr.andb(value, 0wx1) = 0wx1 then`
`  0wx0`
`  else`
`  value`
	`mov  %eax, %edx`
	`and  $1, %edx`
	`cmp  $1, %edx`
	`mov  $0, %edx`
	`cmovne %eax, %edx`
	`mov  %edx, %edx`

Figure 4.3: The two kinds of read-barriers, whose code can be embedded into the skeleton code from figure 4.2. The x86 assembly presented here uses the GAS (GNU Assembler) syntax, which specifies source operand first, then destination, which is opposite of the presentation in [Blac04b]. Notice that the constant -4 is the bit-wise negation of *0wx3*, which is an equally portable way of specifying the same

As mentioned in the beginning of this section, Baker's incremental copy collector is one of the collectors that use read-barriers. Its read-barrier copies (evacuates) objects from one semi-space (Fromspace) to the other semi-space (Tospace), whenever the mutater reads an object still in Fromspace. This is precisely what the function `evacuate` from figure 1.1 in section 1.2.2 does. The way the implementation of that function has been structured in that figure is almost the same as for the presented read-barriers here. The only difference in the function's structure is its parameters, which are tailored to Cheney's copy collector from the figure in which the code is shown, rather than to the general read-barrier presentation framework from [Blac04b] which is also used in this section. Notice how the function otherwise follows the skeleton code in figure 4.2. Notice also that, it is a conditional read-barrier with exactly the same `if`-expression as in figure 4.3. The only part which has been replaced is the part in figure 4.3 that returns zero. This part is the slow path of the read-barrier, which should therefore possibly be generated as a reusable function in the assembly code. This actually also implies that the generated assembly code for the parts which are similar may not be the same, since the value in the assembly code is returned by a conditional move instruction to overwrite the constant zero. Even though the generated assembly code might look different, it still might be as efficient, since there is a conditional instruction of some form in both cases.

The final kind of read-barrier which will be presented is the Brooks-style barrier from [Broo84], which is described in section 8.5 pages 206-207 in [Jone96] and illustrated in diagram 8.10 in [Jone96]. It has been used for turning the conditional read-barrier just described from Baker's algorithm into an unconditional read-barrier, although the resulting method then also requires an incremental-update

write-barrier, as will be explained further in section 4.12.1 about Baker's algorithm. The Brook-style barrier is simply a pointer indirection. This means that instead of reading an object at an address directly, the address is expected to have a pointer to be followed, which points to the actual object being read. Implementing this barrier in Standard ML can be done by the following code:

```
UnsafeMemoryAccess.addrtoword(
  UnsafeMemoryAccess.readaddr(UnsafeMemoryAccess.wordtoaddr(value), 0)
)
```

This code would then have to be inserted into the skeleton code in figure 4.2, where it can be noticed that the calls to `addrtoword` and `wordtoaddr` cancel out, such that the combined barrier-code actually becomes just:

```
UnsafeMemoryAccess.readaddr(UnsafeMemoryAccess.readaddr(slot, 0), 0)
```

As mentioned earlier, this kind of barrier is used in the Metronome system from [Baco03].

4.11.3 Performance of Read-Barriers

As stated in the previous section, read-barriers may be implemented in software by the compiler emitting a few instructions at pointer reads. However, pointer reads constitute 13-15% of all instructions (page 188, section 8.2 in [Jone96], with reference to [Zorn90]). It is claimed in section 8.2 page 188 in [Jone96] that software-based read-barriers are too expensive, but [Baco03] has since shown otherwise, where only 4% overhead on average (10% in the worst case) were reported for the Brooks-style indirection-based read-barriers in their Metronome system. The Metronome system from [Baco03] will be reviewed in more detail later in section 6.7. The article [Blac04b] quite thoroughly dispels the assumptions by researches that barrier overhead should be a motivator for avoiding to use barriers. The measurements in [Blac04b] are direct measurements on the barrier overhead and their methods are able to disregard the impact of the compiler (Jikes RVM) and garbage collector (MMTk [Blac04a]) itself. Section 3.1 in [Blac04b] also describes how they can ignore the cost of remembered sets, by relying on a full heap trace to always infer live objects, whether the remembered sets are used or not. According to their measurements, read-barrier overhead of unconditionally masking out low-order bits of a pointer has from 0.85% overhead on PowerPC up to 8.05% overhead on AMD, whereas [Zorn90] earlier reported up to 20% overhead. As mentioned in section 2 in [Blac04b], the work in [Zorn90] uses untyped Lisp, where pointer versus non-pointer filtering gives overhead in itself, which is not related to the read-barrier overhead. They also state in section 2 in [Blac04b] that the work in [Zorn90] did not account for cache effects, register pressure and branch prediction, as they do. In [Blac04b] they even find that second order locality effects were sometimes more important than barrier overhead, leading to counter-intuitive speedups.

According to section 8.3 page 188 in [Jone96], the expense of read-barriers means that they are rarely, if ever, used with non-moving garbage collectors. An exception is that Baker's Treadmill, which will be investigated in detail later in section 6.6, is a non-moving collector and uses read-barriers, but as will be argued, write-barriers should work with that method as well and may even be more appropriate. The already mentioned Metronome system from [Baco03] is also a more recent exception, since this is a mostly non-moving collector.

4.11.4 Write-Barriers

This section describes write-barriers in more detail. We will see the following examples of the two kinds of write-barrier methods, snapshot-at-the-beginning and incremental-update:

- **Incremental-update barriers**: Steele-barrier [Stee75] and Dijkstra-barrier.

  There is also a four-colour incremental-update barrier by Kung and Song (see section 8.3 page 199 in [Jone96]), but this will not be presented

- **Snapshot-at-the-beginning barriers**: Yuasa [Yuas90]

In snapshot-at-the-beginning write-barriers, whenever a pointer is updated, the object which was pointed to *before* the update is marked as being grey, to ensure that the collector will visit that object again.

In incremental-update write-barriers, potentially disruptive pointer writes are recorded. Whenever a pointer is updated, either the *updated pointer* or the object pointed to *after* the update is marked as grey, depending on the method.

Whichever of these two barriers are used, the greyed cells will be visited by the collector, before the collection can be considered complete. For mark-sweep collectors, for which these barriers are commonly used, the tricolour (or tetracolour) abstraction can be implemented with two colour-bits associated with each memory cell, or with a mark bit and a stack. If using mark bits and a stack, the marked cells are considered black, unless they are also in the stack, in which case they are considered grey. Auxiliary data structures increase the amount of space required by the collector, but they reduce the time taken to mark active cells by e.g. avoiding needless multiple traversals of shared memory objects, not to mention infinite loops for cyclic data structures.

As done previously for read-barriers, write-barriers will be presented in a way similar to [Blac04b]. Four general kinds of barriers will be shown: 1) boundary barrier, 2) object barrier, 3) zone barrier and 4) card barrier. These general barriers focus on the various kinds of performance overhead when implementing the barriers, not how to implement specific barriers for memory-management methods. The source code will be shown as Standard ML code with the corresponding x86 assembly instructions shown. The reader is referred to [Blac04b] for the Java and PowerPC versions.

The skeleton source code in Standard ML for presenting the write-barriers is shown in figure 4.4. This is shown as a function named `writeBarrier`, which *should* be inlined, just as the `readBarrier` function from earlier. The code in figure 4.4 uses the same module *UnsafeMemoryAccess* as described for read-barriers.

The code for the four general kinds of write-barrier implementations are given in figure 4.5, where both the Standard ML code and the 32-bit x86 assembly (IA-32) instructions code are shown. The code from this figure is to be inserted in the skeleton code in figure 4.4, where it replaces the line with the comment. Again, the x86 assembly presented here uses the GAS (GNU Assembler) syntax, which specifies source operand first, then destination, which is opposite of the presentation in [Blac04b]. For the last instruction in the card write-barrier, this difference in representations is even bigger; that instruction means: "move the constant 1 to the address in register EBX plus the constant offset zero plus the offset in reigster EAX multiplied by one". The assembly instructions shown generally do not specify operand sizes, but this particular instruction should probably be `movb`, if it is to write a byte, whereas the other instructions would typically be e.g. `movl`. This is a quite specific barrier operation, shown in Standard ML as the function `setByteAtOffset`, whose definition is not given (neither is it given in [Blac04b]). Since the multiplicative constant in the assembly instruction is one (as is the value written), I would assume that it could be implemented by `UnsafeMemoryAccess.write8(cardTable, card, 0wx1)`. The function `write8` is defined later in figure 8.4 in section 8.3 with the other memory access functions.

```
(* The writeBarrier function should be inlined *)
fun writeBarrier(src : addr, slot : addr, tgt : addr, mode : int) : void =
      let
        val srcValue  = UnsafeMemoryAccess.addrtoword(src)
        val slotValue = UnsafeMemoryAccess.addrtoword(slot)
        val tgtValue  = UnsafeMemoryAccess.addrtoword(tgt)

        (* Replace this line with the write-barrier code *)
      in
        UnsafeMemoryAccess.writeaddr(slot, 0, tgt)
      end
```

Figure 4.4: Write-barrier skeleton code in Standard ML. One of the individual barriers in figure 4.5 are embedded into this by replacing the line with the comment. A loose specification of the module *UnsafeMemoryAccess* can be found later in section 8.3, particularly in the figure 8.4

For the first three barriers shown, the jump instructions with the constant zero in the assembly code are generally what should be replaced with the code to be performed, typically a remembering call in the Standard ML code in these cases.

It is common to combine barriers when forming hybrid garbage collection methods. Therefore, a typical hybrid write-barrier is sketched in figure 4.6. The barrier is a combination of a boundary and an object barrier. Notice that if the barrier is specialized in different parts of the code, depending on what the *mode* value is, the if-expression on *mode* might be possible to optimize away. The boundary barrier is typical in two-space generational collectors, where copy collection is often used for young memory objects (residing in a nursery space) and some other garbage collection method may be used for older memory objects (residing in a mature space). Two-generation hybrid methods were introduced in section 4.10 and many examples of such methods appear in this thesis.

The write-barrier descriptions so far has focused on high-performance implementations. We now turn to actual barriers used in memory-management systems.

The Yuasa write-barrier [Yuas90] is the typical snapshot-at-the-beginning barrier. The barrier traps updated pointers (but not initializing writes) and marks the object pointed to *before* the update as grey. It does this by setting the cell's mark bit and pushing a reference to the old cell onto the marking stack. This method perserves the old object whether or not it is garbage. Snapshot-at-the-beginning are very conservative. No objects that become garbage in one garbage collection can be reclaimed in that same collection; they must at least wait until the next collection. Standard ML code for Yuasa's write-barrier is shown in figure 4.7 (C-code is shown in Algorithm 8.1 on page 190 in [Jone96]). As can be seen on the code in figure 4.7 in comparison to figure 4.5, the implementation is made as an object write-barrier. If the barrier code is specialized to the marking-phase of an algorithm, the if-expression on the *mode* value can possibly be optimized away. The Yuasa write-barrier is used by the Metronome garbage collector from [Baco03], which will be described in more detail later in section 6.7.

Incremental-update write-barriers are normally less conservative than snapshot-at-the-beginning write-barriers. They incrementally record changes made by the mutator to the memory-object graph, rather than simulating a single static estimate of the reachability graph at the start of a collection. They trap any attempt by the mutator to create a pointer from a black object to a white object, shading one of the two involved memory objects grey.

The Dijkstra-style write-barrier adopts the most conservative incremental-update strategy: the white object is coloured grey, when assigned to an object, regardless of the colour of the object assigned to.

Standard ML	GAS x86 Assembly (IA-32)
Boundary write-barrier	

`val () = if slotValue < NURSERY_START andalso`
`  tgtValue  > NURSERY_START then`
`  rememberSlot(slot)`
`else`
`  ()`
	`cmp  $0xa0200000, %edi`
	`jlge  $0`
	`cmp  $0xa0200000, %ebx`
	`jlge  $0`

Object write-barrier	

`val header = getHeader(src)`
`val () = if WordAddr.andb(header, LOGGING_MASK)`
`  = UNLOGGED then`
`  rememberObject(src)`
`else`
`  ()`
	`mov  -8(%edx), %ecx`
	`and  $4, %ecx`
	`cmp  $4, %ecx`
	`jeq  $0`

Zone write-barrier	

`val () = if WordAddr.xorb(slotValue, tgtValue)`
`  > ZONE_SIZE then`
`  rememberSlot(slot)`
`else`
`  ()`
	`mov  %eax, %edi`
	`mov  %edi, %eax`
	`xor  %ebx, %eax`
	`cmp  $0x400000, %eax`
	`jlge  $0`

Card write-barrier	

`val card = WordAddr.<<(srcValue, LOG_CARD_SIZE)`
`val () = setByteAtOffset(cardTable, card, 0wx1)`
	`mov  (0x290279a), %ebx`
	`shr  $10, %eax`
	`mov  $1, 0(%ebx, %eax, 1)`

Figure 4.5: The four general kinds of write-barriers, whose code can be embedded into the skeleton code from figure 4.4. The x86 assembly presented here uses the GAS (GNU Assembler) syntax, which specifies source operand first, then destination, which is opposite of the presentation in [Blac04b]. For the first three barriers, the jump instructions with the constant zero in the assembly code are generally what should be replaced with the code to be performed, typically a remembering call in these cases. The last barrier is quite specific and is fully specified in the assembly code

```
val () = if mode = STORE_WRITE_BARRIER then
           if slotValue < NURSERY_START andalso
              tgtValue  > NURSERY_START then
             rememberSlot(slot)
           else
             let
               val header = getHeader(src)
             in
               if WordAddr.andb(header, LOGGING_MASK) = UNLOGGED then
                 rememberObject(src)
               else
                 ()
             end
```

Figure 4.6: Standard ML code for a typical hybrid write-barrier. This code can be embedded into the skeleton code from figure 4.4. The barrier is a combination of a boundary and an object barrier. Notice that if the barrier is specialized in different parts of the code, depending on what the *mode* value is, the if-expression on *mode* might be possible to optimize away. The boundary barrier is typical in two-generation hybrid collectors, where copy collection is often used for young (nursery) memory objects and some other garbage collection method may be used for older (mature) memory objects. Many examples of such hybrid methods appear in this thesis

```
val () = if mode = MARK_PHASE then
           let
             val header = getHeader(tgt)
           in
             if WordAddr.andb(header, MARK_MASK) = UNMARKED then
               (setHeader(tgt, WordAddr.orb(header, MARKED_MASK));
                push(markStack, tgt)
               )
             else
               ()
           end
```

Figure 4.7: Standard ML code for Yuasa's snapshot-at-the-beginning write-barrier. The code can be embedded into the skeleton code from figure 4.4. As can be seen on the code in comparison to figure 4.5, the implementation is made as an object write-barrier. If the barrier code is specialized to the marking-phase of an algorithm, the if-expression on the *mode* value can possibly be optimized away

In Dijkstra's garbage collection algorithm, from which this write-barrier arises, excplicit colour bits are used in the memory cells, rather than a mark bit and a stack, as was used for Yuasa's write-barrier from before. Standard ML code for Dijkstra's write-barrier is shown in figure 4.8 (C-code is shown in Algorithm 8.4 on page 192 in [Jone96]). As for Yuasa's write-barrier when comparing to figure 4.5, the implementation is made as an object write-barrier. It should be noticed that there is a potential problem with this implementation for parallel collectors, due to the update being done after the write-barrier in the skeleton code in figure 4.4. We shall, however, not delve into concurrency issues in this thesis and the reader is referred to section 8.3 page 192 in [Jone96] for details on this.

```
val header = getHeader(src)
val () = if WordAddr.andb(header, WHITE_MASK) = MARKED_WHITE then
           setHeader(src, WordAddr.orb(header, MARKED_GREY_MASK))
         else
           ()
```

Figure 4.8: Standard ML code for Dijkstra's incremental-update write-barrier. The code can be embedded into the skeleton code from figure 4.4. As can be seen on the code in comparison to figure 4.5, the implementation is made as a simple object write-barrier

The final barrier that will be shown is Steele's incremental-update write-barrier. Rather than painting the white object grey, when assigned to an object, the object assigned to is instead coloured grey. This is a less conservative approach than Dijkstra's algorithm. It may cost an extra visit by the collector to the object being assigned to, but it will reduce the amount of floating garbage left at the end of the collection. The Steele write-barrier is shown as Standard ML code in figure 4.9 (C-code is shown in Algorithm 8.5 on page 193 in [Jone96]). As for the Yuasa write-barrier, the implementation is shown with marking of a mark bit and a stack. The same remark about as for Dijkstra's algorithm applies regarding concurrency and the skeleton code in figure 4.4.

```
val () = if mode = MARK_PHASE then
           let
             val tgtHeader = getHeader(tgt)
           in
             if WordAddr.andb(tgtHeader     , MARK_MASK) = MARKED andalso
                WordAddr.andb(getHeader(src), MARK_MASK) = UNMARKED then
               (setHeader(tgt, WordAddr.andb(tgtHeader, UNMARKED_MASK));
                push(markStack, tgt)
               )
             else
               ()
           end
```

Figure 4.9: Standard ML code for Steele's incremental-update write-barrier. The code can be embedded into the skeleton code from figure 4.4. As can be seen on the code in comparison to figure 4.5, the implementation is made as a simple object write-barrier, as the other write-barriers

4.11.5 Performance of Write-Barriers

Many of the general performance considerations regarding barriers were already described in section 4.11.3 and this section should be understood as an extension of that section.

According to [Blac04b], the card and zone write-barriers have very architecture dependent performance. The zone barrier is among the cheapest for PowerPC but quite expensive for x86. On the other hand, the card barrier is among the cheapest for x86 but quite expensive for PowerPC. The two other kinds of write-barriers and the hybrid write barrier have less architecture-dependent performance. Object barriers are the cheapest, but they are also less precise. The boundary write-barrier is the most expensive. The hybrid write-barrier performs somewhere inbetween the object barrier and the hybrid barrier.

A reasonable generational barrier, which is assumed to be the hybrid barrier, has over 2% average mutator overhead and less than 6% in the worst case. Hence, the performance overhead in using write-barriers is not prohibitive.

Since write operations are more rare than read operations in programs, write-barriers are generally preferred over read barriers. This is especially true for functional programming languages, such as CeXL and Standard ML, which are primarily considered in this thesis. In these languages, write operations can only happen for references and arrays and it is known at compile-time which variables have such types. Therefore, for functional programming languages, write-barriers can likely be tolerated to be slightly more expensive than in imperative languages. This may be a useful observation, but the ideal is of course to keep the overhead of write-barriers low.

4.11.6 Hardware and Page-Protection Barriers

Modern general purpose computers do generally not have specialized support for barriers, as certain older computer architectures did (section 8.2 page 188 in [Jone96]). Hence, the only realistic option for hardware support for memory barriers is to use operating system support, such as page protection, which typically relies on the memory mapping unit, which most computers nowadays do have. Specialized hardware-barrier support for memory protection by the operating system is described in section 8.9 in [Jone96].

Memory-protection-based barriers are, however, generally considered too inefficient, as stated in section 2 in [Blac04b]. The Appel-Ellis-Li collector, which will be briefly reviewed later in section 6.2, relies on memory-protection mechanisms supplied by the operating system. Zorn [Zorn90] suggests that methods based on hardware virtual memory protection can never be competitive to software barriers or special-purpose hardware. His measurements suggest that a memory-protection fault may take up to 10,000 CPU clock cycles, since it is normally designed for irretrievable errors in typical operating systems. This can therefore not be used for real-time garbage collection, since supposedly, no system that relies on such methods can ever provide hard real-time bounds.

As an example from a method which will be central to this thesis, section 3.5 in [Blac08] state that their Immix method uses an efficient object-remembering barrier, as described in [Blac04b], rather than page protection. This is in the part of their method where they use the sticky-mark-bits algorithm from [Demm90] to achieve in-place generational garbage collection, although the article [Demm90] uses page protection and card marking.

4.12 Incremental Garbage Collection

This section introduces those of the concepts of incremental garbage collection which have not yet been introduced.

4.12.1 Baker's Algorithm

One of the early methods, which were interactive and at that time claiming to be real-time, is Baker's algorithm. The method is from Baker's seminal paper [Bake78] and is described in section 8.5 in [Jone96] and in section 13.6 in [Appe98]. Some important information about the method is the following:

1. It is a copy collector with a fine-grained read-barrier, which copies (evacuates) memory objects from Fromspace (white objects) to Tospace (making them grey), before the object's pointer being read by the mutator is returned. This is as the function `evacuate` from figure 1.1 in section 1.2.2, as was also argued in section 4.11.2
2. Objects may also be evacuated lazily, as mentioned further below
3. It has a special Tospace layout, where surviving data is compacted to the bottom, while allocation is made from the top (p. 203 section 8.5 in [Jone96])
4. The algorithm is more conservative than incremental-update write-barriers, but less so than snapshot-at-the-beginning algorithms (p. 203 section 8.5 in [Jone96])
5. The memory-object copying is unchanged from Cheney's algorithm, which was explained in section 1.2.2 and for which code was shown in figure 1.1
6. An amount of space is scanned when the mutator allocates a block of memory, where the amount of scanned space is typically a constant multiplied by the size of the requested memory block

Baker's algorithm is incremental, but does not provide any real-time bounds in many respects. For example, the root set is traced and copied all at once, instead of incrementally, which however is somewhat tackled by scanning a constant multiplied by the size of the requested memory block. The hope in this is that most data will have been copied by the time that the next collection is to be performed. Another limitation is that when incrementally copying objects, large objects consisting of many nested children may take a long time to copy. A possible improvement on this is to copy (evacuate) large objects lazily, using both a forwarding pointer and a backward link for large objects. This gives slightly more space overhead and requires an extra conditional check on a pointer. A final less attractive property of the method, which is inherent to copy collection, is that it needs to copy around all live data, which takes time proportional to not just the number of live objects, but the amount of live data in bytes. This is what non-moving garbage collectors try to avoid.

An optimization to Baker's algorithm which removes the conditional test in the evacuating read-barrier is to use a Brooks-style [Broo84] read-barrier, which was described earlier in section 4.11.2. This works by putting a forward pointer at the header of each memory object. This forwarding pointer then points to where the object is actually located, which may be in either Tospace or Fromspace, depending on whether the object has been forwarded or not. The mutator thus sees objects in both Fromspace (white objects) and Tospace (black or grey objects). To prevent pointers from black to write objects from occurring, a write-barrier becomes necessary, for recording when imperative objects are updated. When such updates are performed, the write barrier must make sure that the object (value) being assigned to the imperative object is evacuated, before performing the write operation. When the object

is evacuated, the indirection pointer is also updated to point to the evacuated object's new location, which is the reason why the read-barrier can avoid the conditional test. The overhead in this method is the extra space for the forwarding pointer, which may be expensive for small memory objects. The overhead of the write-barriers is justified, because write operations are less frequent than read-operations and the read operations become cheaper.

A final optimization worth mentioning for Baker's algorithm is that instead of allocating new objects into Tospace (i.e. as black objects), they can be allocated in to Fromspace (i.e. as white objects). The reason is that the allocated objects will then not be scanned until the next garbage collection, thus allowing newly allocated objects more time to potentially die (point 1 in section 3.3 justifies why this might be a good idea). This in some sense turns the method into a two-space generational collector without any additional overhead.

Although Baker's algorithm does not meet real-time bounds, it still might be good enough for certain interactive settings.

4.12.2 Dijkstra's Algorithm

One of the important incremental mark-sweep algorithms is Dijkstra's algorithm, but there was not enough time to present this, so the reader is referred to section 8.3 in [Jone96] for this.

4.12.3 Generally About Incremental Garbage Collection Methods

The following is a list of some considerations regarding incremental garbage collection methods:

1. Incremental collectors typically need extra head room, compared to their non-incremental counterparts, due to having to make sure not to run out of memory before garbage collection (reclamation) is done (page 184 section 8.1 in [Jone96])

2. Some kind of synchronization between the mutator and the collector is needed to indicate that the connectivity of the data structure has changed (page 185 bottom, section 8.1 in [Jone96]):

   (a) For incremental mark-sweep: multiple readers, single writer synchronization is needed

   (b) For incremental copy collectors: multiple readers, multiple writers synchronization is needed

   (page 186 top, section 8.1 in [Jone96])

   This has to be maintained without introducing too long pauses

3. The requirements for consistency of the memory data structure may be relaxed, by letting the collector work with a *conservative approximation* of the graph of live memory objects (page 186, section 8.1 in [Jone96])

4. Some measures for classifying incremental collectors (page 186, section 8.1 in [Jone96]):

   (a) How conservative are they? More conservative collection gives more floating garbage

   (b) What are the bounds on the pauses? For example, parts of the collection algorithm, such as root set processing, may be uninterruptable. If uninterruptable parts take too long, this compromises the incremental nature of the algorithm

5. Tricolor marking is one way to remember garbage collection progress and to maintain consistency (page 186-187, section 8.1 in [Jone96]). It is used by many methods in one way or another and will frequently be used in explanations

6. Overhead of incremental garbage collection (page 188, the end of section 8.2 in [Jone96]):
   (a) Conservativeness of the barrier used and how it affects the collector's view of reachability of the memory graph may affect performance
   (b) Time and space costs of barriers depend on selectivity (conditional or unconditional), frequency and implementation (colour bits or mark stack)
   (c) Pause times depend on the amount of work done by barriers and on how a collection cycle is initiated or terminated

4.12.4 Concurrent and Real-Time Memory Management is Hard to Do

This section outlines the level of the challenge in turning a memory-management system into a real-time (not just interactive) system or into a concurrent (not just incremental) one.

- **Concurrency challenges**: *Concurrent* (not just incremental) *reference counting*:
  - Shared pointer access is *very expensive* and earlier attempts at making concurrent reference counting were *unsuccessful* (section 8.4 in [Jone96])
  - Turning the Metronome system from [Baco03] into a concurrent system, as in [Auer07], took a large engineering effort, according to [Auer07]
- **Real-time challenges**: Often, systems claiming to be real-time do not actually meet any hard real-time constraints and should therefore probably be referred to as being just interactive

The above considerations hopefully justifies why this thesis does not aim to create neither a concurrent nor a real-time memory management system. We only seek to try to limit the duration and frequency of pauses, in order to get a system suitable for interactive use, without concurrency support. It would be possible to support concurrency in the program without the garbage collector being concurrent with respect to the program, but in general, introducing concurrency complicates things considerably.

Chapter 5

Static-Analysis-Based Memory-Management Methods

The garbage collection methods presented thus far mostly handle memory management at runtime. This section presents some methods, which rely either entirely or partly on static program analysis or types, which are computed already at compile-time.

5.1 Region Inference

Region inference is a memory management method, which relies solely on static program analysis and runtime arena allocation into regions of memory, where an entire region at a time is freed at statically computed program points. The method was introduced in the seminal paper [Toft94] and has since been revised and improved in several articles, e.g. [Toft97]. Importantly, it has been combined with Cheney-style copying garbage collection, extended for garbage collecting individual regions, in the ML Kit compiler [MLKit]. This is presented in the article [Hall02], which is based on the work in [Hall99].

5.1.1 Region Inference is Not a Comprehensive Memory Management Solution

Unfortunately, region inference in itself is often not sufficient in itself for memory management, since it is not possible to statically analyze precise enough memory usage for all programs. This is evidenced by the article [Hall02], where region inference only reclaims the ideal 100% of all memory for a few of the benchmark programs (section 4.3 in [Hall02]). Section 4.5 in [Hall02] states that, when bootstrapping the ML Kit itself, there is one big region, which would grow infinitely, if garbage collection hadn't reclaimed memory in this region. According to a private communication with Niels Hallenberg [Hall10], the ML Kit was not able to bootstrap itself within the 1 GB of memory that their machine had at that time, which means that the addition of garbage collection was necessary, in order to be able to bootstrap the ML Kit with itself at that time. Also according to this communication with Niels Hallenberg, it is possible to write very small programs, for which region inference fails to prevent memory from growing infinitely and eventually exhausting the heap. This also seems to be the case for some of the benchmark programs in table 3 in section 4.3 in [Hall02], e.g. particularly the program `logic`, for which only 0.1% of the memory is reclaimed by region inference and 99.9% is reclaimed by garbage collection.

It is stated in section 1.2 in [Hall02] that, code modification is usually required to get good behaviour for region inference in practice. They state that, it is usually very few lines of code, examplified as being only 10 lines of code for a 60,000 line program called Anno Domini. The problem here is to find out

which lines of code needs modification, which requires fairly detailed knowledge of region inference. The ML Kit comes with a profiling tool, which may be helpful for locating such issues, but it does not directly solve the problem. It is problematic if the memory usage grows infinitely for interactive programs, which typically have some kind of main loop, which runs continuously, until the user quits the program.

Back in 2001, I wrote two small computer games, "The Unlimited Game" and "ABC Expedition", in Standard ML, where I tried compiling one of them with the ML Kit. This game ran out of memory after running for a while, meaning that region inference was not a feasible solution in this case. The released versions of the games were compiled with the MLton compiler [MLton] and they are still available for free download [HcPE01]. The games do not run out of memory when compiled with the *MLton* compiler, but on the other hand, the games then have occasional garbage collection pauses, which is particularly evident in the game "ABC Expedition". This is one of the reasons why this thesis focuses on memory management methods suitable for interactive applications. It is necessary, if languages like Standard ML are to be used for game development or for performing time-critical tasks.

5.1.2 Advantages of Region Inference

From reading the preceding section, one might get the impression that region inference is not very useful, which is however not the case.

One benefit of region inference is that it can seriously reduce, possibly almost entirely eliminate, the pauses typically associated with garbage collection, which makes it a very relevant to consider for interactive applications. However, if it trades pauses for memory leaks, as argued in the previous section, this is a serious problem.

In section 4.1 in [Hall02], it is shown that tagging in isolation adds substantial time and space overhead, close to 50% overhead on memory usage. An important benefit of region inference is that it makes tagging unnecessary.

For the comparisons with garbage collection in that article, they add tagging to their region inference, both to make a fair comparison and to allow combining region inference and garbage collection. In section 4.2 in [Hall02] they argue that, when enabling region inference with garbage collection, as opposed to only using garbage collection, there are three consequences: 1) the number of collections decrease dramatically, 2) execution times usually decrease and 3) space-usage depends highly on the program. They also show that using only regions without tagging is faster than using regions with tagging and garbage collection, thus showing that region inference offers the memory management strategy with the fastest execution times. In this comparison it should however be noted that, their garbage collection method, Cheney-style copy collection, is not among the most efficient garbage collection methods, so it is not clear whether or not a purely high-performance garbage collection method could be even faster. It is also in the case of using only region inference that the programmer is required to modify the program, in order to avoid memory leaks. They state as future work in [Hall02] that they would like to combine region inference with tag-free or nearly tag-free garbage collection, which has actually since then been implemented in the ML Kit [MLKit].

5.1.3 Conclusions of Region Inference

As already argued, region inference in itself cannot be considered a comprehensive memory management method. This is the primary reason for not considering it for implementation.

The article [Hall02] suggests for future work that region inference may be used as an advanced escape analysis. This would be useful for limiting the amount of heap allocation, by replacing it with stack

allocation where possible. However, using a simple and more standard escape analysis would likely be much simpler to implement. It should also not be too hard to extend a standard escape analysis with an analysis of possible call-paths for functions, in order to also stack allocate some of the objects, such as records, returned from non-recursive functions. This may not make as much stack allocation possible as with region inference, but it should get part of the way in that direction.

An additional complication in implementing region inference is that it requires a specialized intermediate language, containing language constructs like `letregion` and region annotations. This would seem to increase the general effort of implementing a compiler with region inference. If incorporating region inference in a compiler is desired, I would propose investigating if it would be possible to formulate the region inference as an algorithm using more standard compilation techniques, such as symbol tables, without any special language constructs in the intermediate language representations. Such an investigation might be a considerable project in itself. I have not investigated whether formulations of such algorithms already exist, but this could be the case.

5.2 Typing Via Static Capabilities

The article [Walk00] presents an alternative kind of static memory analysis framework to region inference, known as a static capability calculus. This uses continuation passing style (CPS) rather than the stack-based style of region inference. The article [Walk00] presents both kinds of calculi and shows how to translate from the stack-based region calculus of [Toft94] to the continuation passing style (CPS)-based capability calculus.

An important difference between the two calculi is that, the capability calculus dynamically manages regions by a `newregion` (for creating a new region) and a `freeregion` (for freeing a region and all memory allocated in it) to manage memory, whereas the region calculus uses a `letregion ... in ...`-construct, which statically defines a lexical scope for a region, which is freed when that scope is left. In both calculi, the program may allocate into live regions using the notation `v at r`.

It is argued in [Walk00] that region inference may cause disasters when used with continuation passing style (CPS). It is also argued that proving safety properties is easier with the capability-style calculus than the stack-based region calculus. Finally, it is shown by example that the region calculus may give unnecessarily long region life-times, although optimizations for stack-based region calculi exist to improve on this.

Their translation from the stack-based region calculus into the capability calculus not only translates the memory calculus, but also transforms the program into continuation passing style (CPS), where they have a nice way to avoid creating too many "administrative" continuations. The existence of region *inference* algorithms means that this inference can be used as a front-end to a capability-style calculus, for which there did not exist any region inference algorithm at that point; I have not investigated whether or not one exists now. They describe an optimization, which is able to allow deallocation of heap-parameters in tail-recursive functions in each iteration. This is not possible with the lexically scoped region life-times of their region calculus.

I wrote a small report [Anoq04] back in 2004, which summarized the article [Walk00] in more detail and compared it to [Wang00]. The calculus in [Wang00] uses a *region passing* semantics, where regions are passed around as values at runtime, which is a novel approach of that article. It has a penalty at runtime, but can free up more memory in some cases, compared to what purely static systems like the one presented in [Walk00] can do. The methods from [Wang00], and particularly the later Ph.D. thesis [Wang02], will be reviewed in more detail in the next section.

5.3 Combined Methods for Managing Memory with Types

The Ph.D. thesis [Wang02] presents methods for doing typed memory management using various combinations of techniques, when preserving types during compilation. In particular, they implement the garbage collector as part of the compiled program and obtain guarantees that the memory management is safe. They use a capability-style region calculus for their static analysis, presented in Chapter 3 of [Wang02], rather than a type-and-effect-style calculus, as in the region inference from earlier and from [Toft94]. With their capability calculus, they are able to give memory safety guarantees, for example when making a flip between the two semi-spaces in a copy collector.

They show how to implement a simple copy collector in Chapter 4 of [Wang02] and explain the challenge of sharing between memory objects, where sharing is destroyed if the copy collector blindly performs a deep copy operation on the entire heap. Also, such a deep copy operation would never terminate, if used on cyclic data structures. They solve the problems with handling sharing in Chapter 5 in [Wang02], by using forward pointers within their region framework. In Chapter 6 in [Wang02], they evaluate their methods when integrated into an older version the MLton compiler [MLton], at the time when it used continuation passing style (CPS), which the later versions of [MLton] do not. They compare with MLton's native garbage collector and their results seem good, but there are several factors in play and the comparison is not accurate. They also admit that there is room for improvements on some of the overhead in their prototype.

Chapter 7 in [Wang02] compares heap versus stack allocation of returns versus continuations. Among other things, it is described how return continuations form a list structure in memory and how it can thus be unrolled into a stack with dynamic reallocation. Linear types are presented for handling control of allocation of return continuations. They also present a linear assembly language for which they introduce stack allocation, first-class continuations and compilation of exceptions.

The technical report [Wang00] is earlier than the Ph.D. thesis [Wang02] and contains similar material.

Since the compiler implemented for this thesis does not use continuation passing style, where some good reasons for avoiding this were given in the report [Anoq10a] (to appear as [Anoq10b]), the methods relating to continuation passing style are not particularly relevant for this thesis and will not be considered any further.

5.4 Tag-Free Garbage Collection Using Explicit Type Parameters

The report [Tolm94] shows one way of achieving tag-free garbage collection. A typed intermediate representation is defined for a language like ML. It is an explicitly typed 2nd order λ-calculus. The type information is compiled to the final program where it is used by the garbage collector to avoid boxing data. The places were the type description is a challenge at runtime is at *polymorphic type variables* and *closures*. If the types can be allocated statically, it is not inefficient to allocate it. However, if the types need to be dynamically allocated as data structures to be passed around at runtime, which is usually required in connection with recursive calls and closures, then it gives *further* garbage collection work. Some of the overhead should be possible to avoid by adding certain optimization passes to the compiler.

The compiler implemented for this thesis eliminates polymorphism in one of its first passes, the monomorphization pass, as does the compiler MLton [MLton]. This avoids having to deal with polymorphic types at all during garbage collection and at runtime. There was quite an extensive list of reasons in the report [Anoq10a] (to appear as [Anoq10b]) arguing why one would want to avoid having polymorphic types during compilation and at runtime; the results of the report [Tolm94] is yet another

reason. The implemented compiler also eliminates higher-order functions by closure conversion and there is no particular inefficiency in performing recursive calls. It will be described in more detail later in section 8.2.1 how closures are handled for recursive calls in the implemented compiler. This was actually not properly described in the report [Anoq10a], but it will be in [Anoq10b]. Thus, with the combination of methods used in the compiler for this thesis, no extra work is introduced in using types in the compiler. The methods in the report [Tolm94] are therefore not needed for this compiler, even when letting the garbage collector take advantage of or even rely on having types available for all data.

5.5 Tag-Free Garbage Collection

The previous sections mentioned tag-free garbage collection. This section clarifies the various kinds of tags which can be referred to. Usually, not all tags can be removed.

- **Pointer-identification tags**: For a copy collection algorithm, or even just for the part of copying the graph of memory objects (evacuation), which is used in many methods, it is necessary to know which parts of memory objects are pointers. With static type information, pointers in memory areas can be statically known in advance. This can avoid tagging objects in memory according to whether they are pointers or not. When referring to tag-free garbage collection, it is often primarily these tags which are referred to, secondarily the tags described next, which usually can only be partially avoided

- **Marking tags**: It is necessary to somehow keep track of which objects have already been copied or marked, when tracing memory. Failure to do this results in the copying of cyclic data structures to never terminate. Keeping track of this is usually handled by overwriting the first word of a copied object with a forwarding pointer, for copying collection, or by marking a bit in the object or an object bitmap, for mark-sweep-like methods. This, however, in turn requires being able to determine that a value is marked or that it contains a forwarding pointer, not just e.g. some integer which happens to look like a forwarding pointer. For handling this, it would seem that tagging still becomes necessary. There is, however, one possible solution to this:

  - Divide the heap into two memory spaces: 1) a space of leaf node memory objects and 2) a space of non-leaf node memory objects. I have even seen someone on the garbage collection mailing list (gc-list) state something along the lines of: "Dividing into leaf-nodes versus non-leaf nodes is the most important part. Everything else is trivia" (I cannot recall where). If this division is performed, it becomes easy to distinguish forwarding pointers in the memory area with non-leaf nodes, since all these objects already contain at least one pointer. The memory layout can be designed such that child pointers always reside in the beginning of each memory area, thus making it possible to always overwrite the first pointer in a memory area with the forwarding pointer, when the object is copied. Pointer-values can also generally have their low-order bit used as a tag-bit, if pointers are aligned to even addresses.

    For the leaf node memory objects, one can consider to live with the leaf nodes not maintaining sharing, by copying the leaf nodes as many times as necessary. This will not create any problems with cyclic data structures, since leaf nodes can never be part of a cycle. Alternatively, some kind of tagging or marking can be used, but now only for the memory space with leaf nodes. A separate mark bitmap for this area can be used, if one wishes to avoid changing the memory representation of leaf node values

For the sake of completeness, a third kind of tags, in addition to pointer-identification tags and marking tags, are the tags used in tagged-sum types in many programming languages. These tags are part of the language's semantics, not part of memory management. They can generally not be removed, except sometimes in places where the program can be optimized such that it becomes statically known which type of tag is used at that point.

A general conclusion here is that, static analysis of programs can be used to eliminate many kinds of tags, but not necessarily all. Normally, at least the tags used in marking or otherwise tracing cyclic data structures have to be used.

Chapter 6

Advanced Garbage Collection

This section describes some specific advanced garbage collection methods most of which are very relevant to consider for implementation for this thesis, since they represent the state of the art in important aspects.

Some of the methods, however, particularly the first ones presented, are not good enough for what this thesis is aiming at, at least not without extensions and in comparison to what other methods are available. These methods are mostly presented, because they are either well-known or commonly referred to by other methods.

6.1 Incremental Incrementally Compacting Garbage Collection

The incremental incrementally compacting garbage collection method was originally published in [Lang87] and is presented in section 6.3 on pages 127-128 in [Jone96]. This method is mostly only included because the idea is an interesting example of a hybrid method.

This is a hybrid method where most of the heap is managed by a mark-sweep collector, while being divided into sections. A window of two, usually consecutive, sections are treated as two semi-spaces for copy collection. The copy collection compacts and defragments the heap in sections. At each collection, both the copy collection of the two selected sections and the mark-sweep are done at the same time. This is illustrated in figure 6.1.

They suggest that the method may be combined with an incremental mark-sweep collector, but according to page 128 in section 6.3 in [Jone96], they give few details and I have not looked into this. The method does not otherwise seem incremental and it still seems to have time-complexity of the size of the heap, as a normal mark-sweep. Also, the method would seem to be copying memory in the entire heap continuously, which likely gives overhead for long-lived objects.

The article has no comparisons with other methods, nor are there any comparisons of it with other methods in [Jone96]. I have also not found any newer methods comparing themselves to this, so it is hard to say how well it performs, compared to other methods, but it does not seem suitable for interactive or real-time purposes, unless it could at least be made incremental somehow and even if so, the continuous copying does not seem ideal.

6.2 The Appel-Ellis-Li Collector

The Appel-Ellis-Li collector is presented in section 8.6 in [Jone96] and was originally published in the article [Appe88]. The method relies on memory protection mechanisms supplied by the operating system.

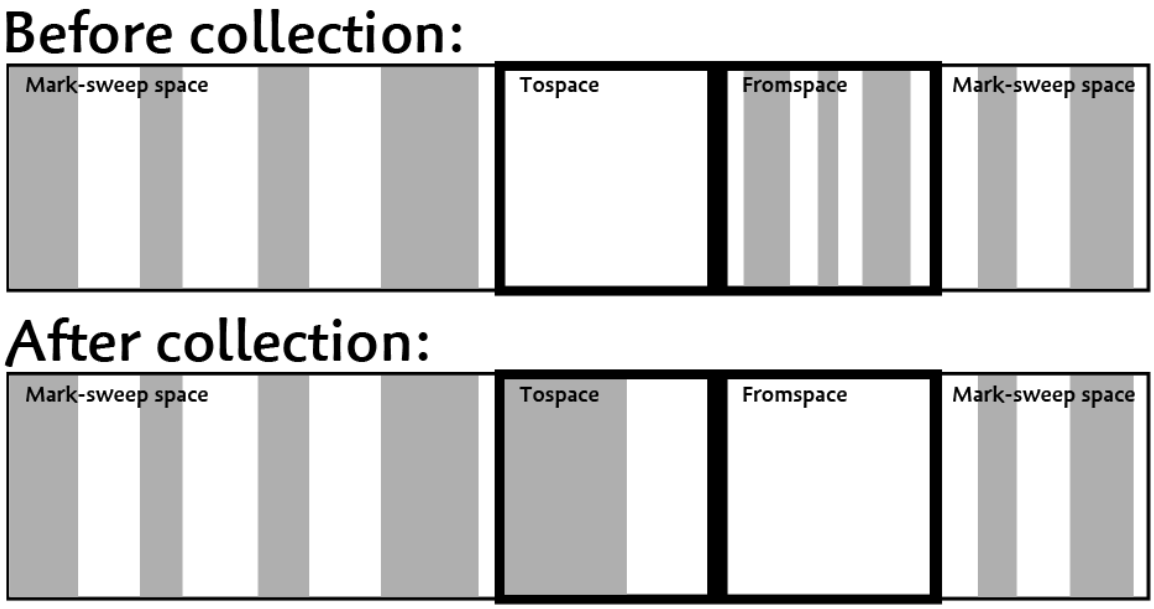

Figure 6.1: Incremental Incrementally Compatcing Garbage Collection. This figure illustrates data areas in the heap and the colours do not follow the colour coding of the tetracolour abstraction

As was argued earlier in section 4.11.6, memory protection has severe overheads, up to 10,000 CPU clock cycles. This collector is therefore not a real-time garbage collection method, since supposedly, no system that relies on such methods can ever provide hard real-time bounds. It will therefore not be considered as a method for implementation in this thesis (see goal number 2 in section 1.3). It is presented here mostly since some methods refer to it, though mostly its performance. Readers who find the details of this method uninteresting may therefore *safely skip the rest of this section*.

The method can be summarized by the following:

1. It is a generally portable and incremental algorithm, supporting concurrency without fine-grain object locking
2. It is based on the copying from Baker's algorithm, but uses a page-wise black-only read-barrier, supported by the operating system's memory protection facilities
3. The mutator is only allowed to see black objects, which is ensured by using a page-based read-barrier for blackening grey objects when they are read
4. For a sequential implementation: the collector removes the protection from the page that caused the page-fault, evacuating their Fromspace children into Tospace pages, which are then protected
5. The garbage collector also scavenges (copies to Tospace) grey pages in the background when handling page faults
6. For a sequential implementation: scanning is done when allocating memory, as in Baker's algorithm
7. For a concurrent implementation: a separate thread scans a complete page at a time (like the read-barrier). When all pages are scanned, the scanner thread blocks until a function `flip` signals that more pages should be scanned

8. The function `flip` must ensure that scavenging is completed (i.e. no more grey pages left) before swapping the semi-spaces. In a concurrent implementation: it must also halt all mutator threads before the flip and the flip must must restart mutator threads and the scanner thread

9. Notice that the memory trap thread and the scanner threads must have access to the protected Tospace pages, without triggering page faults. This can be accomplished by using user mode versus kernel mode on most architectures

The method can be improved further. A summary of its suggested improvements is given here:

1. The global memory lock, which is locked when scanning a page and when allocating memory, is a bottleneck for concurrent systems. This can be improved by using a two-stage collector, which grabs the lock to allocate a *chunk* of memory, where further allocation can be made in the chunk without locking

2. The flip latency can be high, if there are a large number of roots (e.g. many threads or threads with a large stack). These two issues can be improved as follows:

   (a) Handling large stacks: the stacks can be divided into pages and scanned incrementally when the mutator accesses them

   (b) Avoiding to scan the registers of each thread: let the `flip` function set the program counter to a subroutine which saves its own registers

3. Large objects can be handled by a "crossing map" array (see also "card marking" in section 7.5 of [Jone96]). Copying large arrays can be avoided by using back-pointers, as in Baker's algorithm 4.12.1

4. Page scanning may give long pauses. This can be improved by using the extended Baker-Steele lazy copying due to Johnson [John92]. However, this has a greater total cost than the eager version of copying

5. Add generational collection: this improves the performance for programs with larger amounts of long-lived memory cells

The performance of this method is hard to measure, as it depends on the hardware memory-protection mechanisms. The advantages of this method is mostly its portability and its multi-thread support, though, as stated in the beginning of this section, it can never meet hard real-time constraints.

6.3 Nettles and O'Toole's System

The system by Nettles and O'Toole is presented in section 8.7, pages 214-215, in [Jone96] and was originally published as [Nett93]. The system can be summarized as follows:

1. It is an incremental or concurrent garbage collection method for Standard ML, which aims to reduce the garbage collection pauses in general and in particular the cost of synchronization (page 213 in [Jone96])

2. It is based on a copying generational garbage collector, the one from [Appe89], and they incorporate it into the Standard ML of New Jersey (SML/NJ) compiler

3. It does not rely on expensive read-barriers

4. The mutator is allowed to access the original Fromspace objects

5. When copying is complete, the garbage collector replaces the mutator's roots with pointers to their Tospace replicas and then discards Fromspace

6. The method requires non-destructive copying, e.g. by storing the forwarding pointers in an extra word for each memory cell, possibly in an overwritten header word, where a forwarding pointer check is made whenever the header is accessed. This indirection should only give a performance penalty in their compiler for polymorphic equality and a few other type-specific operations

7. When modifications are done to objects in Tospace, which have already been copied from Fromspace, they are recorded in a "mutation log" by the mutator (an example of relaxing the consistency requirements). Modified and scanned replicas must also be re-scanned, to ensure that new children of the objects are copied

8. The mutation log can be maintained by a write-barrier, rather than the more expensive read-barriers of the Baker and the Appel-Ellis-Li methods. This is useful for functional languages like Standard ML and CeXL, where destructive updates are rare

9. Collection is complete when there are no more unscanned objects in Tospace and the mutation log is exhausted

10. The method is well-suited to generational collectors, where the write-barrier can record inter-generational pointers and mutations

Their method achieves good performance for Standard ML. The overall slowdown is less than 20%, typically less than 10%, if the incremental techniques were restricted to the major collections. The time spent on minor collections increased, when using this method. The method is well-suited for execution in a separate thread, since it requires little low-level synchronization. Most pause times were around 5ms on their machine with four 33Mhz MIPS R3000 processors.

6.4 The Huelsbergen and Larus Collector

This method is presented on pages 215-216 in section 8.7 in [Jone96]. It is similar to the system by Nettles and O'Toole described in the previous section, but their methods were used without generational collection. Incorporating the generational collection from [Appe89] might have improved their performance and made a more fair competition with Nettles and O'Toole's system.

The method relies on many objects being immutable, which is a reasonable assumption for languages like Standard ML. It lets the mutator access immutable data without impediment, but uses only the Tospace copies of objects for mutable objects. The method requires an extra check for mutable data.

The pauses were never more than 20ms on their machine. The elapsed times were also significantly shorter than for Appel's copying generational garbage collector from [Appe89].

6.5 The Doligez-Leroy-Gonthier Collectors

The Doligez-Leroy-Gonthier collectors are presented on pages 216-218 in section 8.7 in [Jone96]. There are two different articles published about this, for two different versions of the collector, the articles [Doli93] and [Doli94]. The methods are summarized here:

1. The method uses two generations, a young (nursery, although they do not use this term) generation per mutator thread and an old (mature) generation, which is shared among all threads

2. Mutable objects are only allocated in the second (mature) generation

3. Copying collection is done for the young heaps, with survivors being promoted to the shared heap

4. The copying collection only stops the thread involved and neither requires synchronization with other threads nor with the old (mature) shared generation

5. A dedicated thread collects the mature shared heap concurrently with the mutator threads and their minor collections

6. The concurrent collector cannot move objects in the shared mature heap, since mutator threads might hold references to them, so it uses mark-sweep collection based on Dijkstra's algorithm, where the free list is coloured a fourth colour, in order to improve the marking efficiency

7. Updates to mature objects, which makes them point to young (nursery) heap objects, triggers the young object and its decendants to be copied into the mature heap. This does not cause problems for immutable objects. Such copies leave a forwarding address behind in the header word of the young heap, to assist the next minor collection

8. Their concurrent collector in [Doli94] avoids the cost of the write-barrier for local variables, but it requires a fairly complex protocol for initiating major garbage collection cycles

Most minor collections completed in less than 10ms with their system, but this is when using only 32kb young heaps. The major collection load per mutator was below 5% (suggesting that it may be usable with up to around 20 threads), but this is for a byte-code interpreter (Caml Light) running 4-8 times slower than a natively compiled mutator (and thus likely has a slower rate of memory allocation). Mutable data must be allocated in the shared heap, whether or not it is shared, which may be expensive. Allocation in large chunks can improve but not eliminate this overhead. Assignment of a pointer from an old (mature) object to a young object requires the transitive referential closure of the young object to be copied to the shared mature heap and the cost of this copy cannot be bounded.

6.6 Baker's Treadmill Collector

Baker's Treadmill collector is from the article [Bake92] and is also presented on pages 218-220 in section 8.8 in [Jone96]. Figure 6.2 is a visual illustration of the method, which is summarized here:

1. It is a non-moving garbage collector, which is advantageous, especially for incremental garbage collection, but also with respect to compiler optimization and efficient multi-threading support

2. The heap is divided into four coloured sets: 1) scanned objects (black), 2) visited but unscanned objects (grey), 3) objects not yet visited (white) and 4) free space (off-white). In comparison, the semi-space heap arrangement, as used in copying collectors, is just one but not the only way of representing these four sets

3. This method organizes the four sets in a cyclic doubly-linked list, "the treadmill", with the four colour segments arranged contiguously in the list (but not necessarily contiguously in memory), delimited by four pointers: `free`, `B`, `T` and `scan`

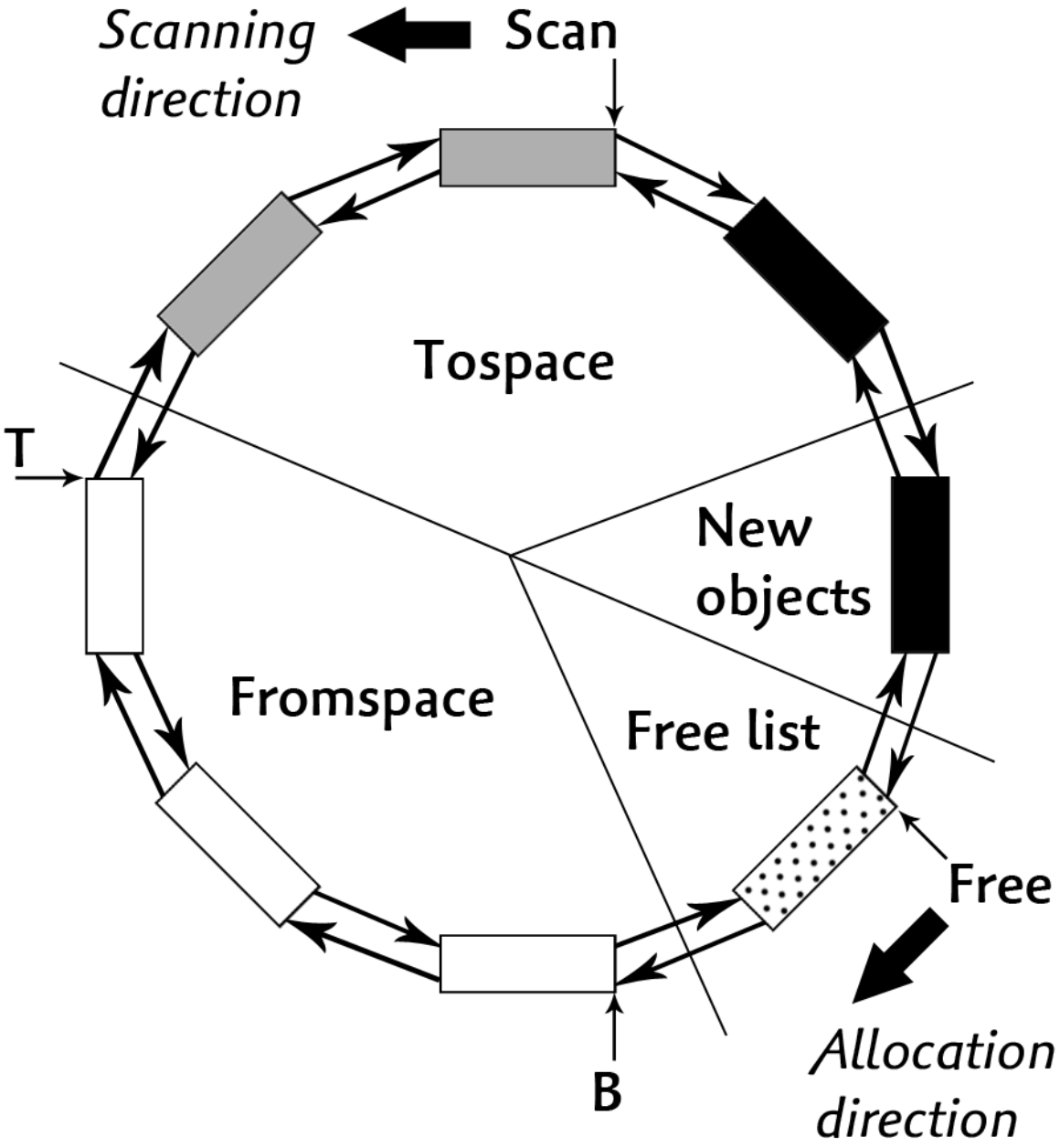

Figure 6.2: Baker's Treadmill collector. The figure illustrates memory cells, colour coded with the tetracolour abstraction and connected into a doubly linked list. The four pointers *free*, *B*, *T* and *scan* are shown, along with the allocation and scanning directions

4. Allocation is done by simply advancing the `free` pointer clockwise around the threadmill. Marking is mentioned to be equally simple in section 8.8 in [Jone96], but it is not described. It seems that it should be done by performing the first level of root set tracing, followed by scanning, as described next

5. Scanning is done by advancing the `scan` pointer anti-clockwise. When a grey cell has been scanned, the scan pointer is moved, in order to paint the grey cell black

6. No explicit representation or manipulation of colour bits is necessary for the operations so far

7. If a scanned pointer refers to a black or grey cell, no action is taken

8. If a scanned pointer refers to a white cell, that cell must be unsnapped from the white segment and snapped into the grey segment. Snapping and unsnapping is a constant-time operation and offers the algorithm the potential to meet real-time bounds. This is the only point at which colours need to be distinguished, so only one colour bit (white vs. non-white) is needed per cell

9. The snapping mechanism allows a choice of traversal strategies, since white cells can be added to any of the two ends of the grey segment. If a cell is snapped between the white and the grey segment (at the `T` pointer), the traversal is breadth-first (as in a traditional semi-space copy collector). Snapping in cells between the grey and the black segment gives a depth-first traversal. Many authors have observed that depth-first traversal causes fewer cache and virtual memory faults, as described previously in section 4.9 (see also e.g. section 6.6 in [Jone96]). Notice that the depth-first traversal is thus achieved without any use of an auxiliary stack, as is commonly required. In terms of space overhead, the permanent list-links of the treadmill are, however worse, but at least the algorithm guarantees that it will not run out of space during garbage collection

10. A garbage collection cycle is complete when there are no grey cells left, i.e. with `scan` meets `T`

11. When `free` meets `B` (i.e. the free list becomes empty), at which point only white and black cells exist, it is time to flip

12. A flip is performed by reinterpreting the black segment as white and the white segment as off-white. This is done simply by exchanging the `T` and `B` pointers. Thus, the treadmill advances its segments, hence its name

It is mentioned on page 220 in [Jone96] that the method, as described in [Bake92], uses read-barriers, but that since data is never moved, write-barriers are likely to be better, for the following reaons: 1) it should give better performance, since writes are typically more rare than reads, 2) it integrates better with generational garbabge collectors and 3) it may offer better opportunities for optimizations. Incremental-update write-barriers (as defined earlier in section 4.11.1) are used by [Wils93], where the write-barrier can be configured to use either Dijkstra-barriers or Steele-barriers (according to [Wils95]). [Henn93] has an implementation based on a snapshot-at-the-beginning barrier. The performance of these two implementations are however stated in [Jone96] page 220 to be disappointing, e.g. still only somewhat better than deferred reference counting, but that this may be due to the barriers being implemented with smart pointers. The barriers implemented as described earlier in section 4.11 and in [Blac04b] ought to give good performance.

The treadmill method has the following time and space performance properties:

1. It is expensive in space, compared with other non-moving collectors, because of its links, although this is somewhat outweighed, since no additional space is needed for marking. Its space usage is, however, not any worse than copy collectors, since no semi-spaces are used and the links take up no more memory than two copies (corresponding to two semi-space copies) of a typical cons-cell. The overhead of the links is thus lower than a factor of two for objects larger than cons-cells

2. Allocation is more expensive than pointer-based bump allocation, but cheaper than manipulating a linked list (e.g. a free list) or lazily scanning a bitmap

3. Resnapping an object into the grey segment may be more expensive than copying list cells, but less expensive for large objects, since it takes constant time

4. The time required to reclaim white cells is constant: garbage cells are untouched

The challenges or limitations of the treadmill method are the following:

1. The method, as described so far, does not handle objects of varying sizes. [Jone96] page 220 mentions that the references [Bren89], [Whit90] and [Bake85] suggest ways of reducing the costs of manipulating objects of different sizes. For the treadmill method, [Wils93] handles this in their real-time garbage collector by rounding up object sizes to the nearest power of two and using a separate treadmill for each class size. When using multiple free lists, the free lists will not become empty simultaneously, hence, reclaimed cells must be recoloured explicitly, which can be done lazily. It also becomes necessary to distinguish between white and off-white cells

2. A disadvantage of the method, which is not mentioned in [Jone96], is the problem of fragmentation, since objects are never moved. When size-class segregating into multiple treadmills, as described above, this mostly solves the problem. However, using size classes of powers of two gives high internal fragmentation, as described earlier in section 4.4.3 and mentioned in [Baco03]

6.7 Metronome: Bacon, Cheng and Rajan's Real-Time GC with Low Overhead and Consistent Utilization

The article [Baco03] presents a garbage collector, which is referred to as Metronome in later articles, specifically in [Auer07]. The system in [Baco03] solves many of the limitations of earlier garbage collectors. Problems that they solve are a) fragmentation, b) high space overhead, c) uneven mutator utilization and pauses and d) inability to handle large data structures. It is, however, indirectly (in section 2.3 in [Baco03]) suggested that the methods of [Nett93] and [Chen01] might be good enough for mostly functional programming languages, such as Standard ML.

Their method is an incremental uni-processor collector, using a hybrid of a non-copying mark-sweep collector (for handling the common-case collection) and a copy collector (when fragmentation occurs). Some of the specific methods used are the following:

- Segregated free lists, with increases at a rate of $1 + \frac{1}{8}$ between size classes, since a power-of-two would give high internal fragmentation

- The collector is mostly non-copying

- Defragmentation, achieved by copying memory objects of a fragmented page to another page

- Brooks-style [Broo84] read-barriers, which were described in detail earlier in section 4.11.2. These barriers are used when relocating objects
- An incremental snapshot-at-the-beginning mark-sweep collector, similar to [Yuas90], meaning that it uses Yuasa's write-barrier, as described earlier in section 4.11.4
- Arraylets, for avoiding fragmentation and to avoid large objects, which cause several problems (large overhead for copying, no bounds on copying times for incremental garbage collection and they may be impossible to allocate with a fragmented heap)
- Stacklets are suggested, but not used in their implementation
- Various optimizations on arraylets and null-pointers
- A separate heap without read barriers for runtime immortal memory objects

They have both time-based scheduling, which performs well and meet the real-time demands, as well as work-based scheduling. They use polling for the time-based scheduling, since their operating system's timer granularity is only down to around 10 ms. Both their analytic results and evaluation results show good results for time-based scheduling, which is also what makes their method avoid uneven mutator utilization. They find that, for work-based scheduling, it is hard to derive collection-time bounds with closed-form solutions and that it becomes the programmer's responsibility to allocate memory sufficiently discretized and evenly spaced over the program execution time. Often, earlier garbage collectors use some kind of work-based scheduling and they point out a flaw in Baker's definition of real-time in his seminal paper [Bake78], since this is closer to work-based scheduling. They recommend time-based scheduling.

They compare their space overhead with other collectors and find that they often have significantly lower overhead than semi-space collectors and much less fragmentation than pure non-copying collectors.

They state that the reference [Zorn90] finds that software read barriers are much better than protection-based read barriers, but that they still cost around 20%. They get significantly better read barriers than previous results: 4% geometric mean, with worst expected being 8% (but is actually 10% in one benchmark, due to a bug in their system, see section 7.1 in [Baco03]). This also better matches the performances measured in [Blac04b], as was described earlier in section 4.11.

They conclude that their system has highly predictable mutator utilization rates with highly stable pause times at real-time resolution. They guarantee 45% minimum mutator utilization (MMU) (over a 10ms time window) with only 1.6-2.5 times the memory usage of the actual memory high water-mark of the application.

The article [Auer07] extends the garbage collector from [Baco03], and presents it in the context of a complete Java Virtual Machine (JVM) product, including compilation, memory management and execution of programs. They have extended it from the uni-processor system not supporting all Java's RTSJ features, as presented in [Baco03], into being a symmetric multi processor (SMP) system supporting all RTSJ features and having the worst-case latencies reduced by an order of magnitude, thus guaranteeing 67.7% minimum mutator utilization (MMU) (over the same 10ms time window). The system in [Auer07] thus achieves real-time garbage collection with sub-millisecond worst-case latencies. One of the methods they use for achieving this is to use overclocked scheduling, where the collection is split into smaller evenly spaced and more frequent collection quanta, each one around 500μs.

They state that a significant amount of engineering was required for extending the system from a uni-processor to an SMP system, but that this task otherwise used mostly previously known methods. The have also extended the root processing, such that it incrementally processes some number of thread

stacks per collection quantum, thus allowing large numbers of threads. For this, they use an overinclusive approximation to a snapshot, which they call a "fuzzy snapshot", which uses a double write-barrier; an extension to a standard Yausa write barrier [Yuas90]. They also mention several other challenges in scaling their system to a production system, such as handling special Java features, e.g. JNI global references, and soft and weak references, handling Non-Heap Real-time Threads (NHRTs), assigning priorities to collector threads and using their arraylets with the existing Java class library APIs, some of which implicitly assume contiguous memory layouts of arrays.

All in all, the system in [Auer07] is very impressive and seems like the state of the art in real-time garbage collection. They mention outperforming some other competitive systems, such as [Clic05], most of them being significantly outperformed on various differing criteria.

6.8 Mark-Sweep with Parallel Incremental Compaction

In the article [BenY02], an existing mark-sweep method is extended from supporting only full-heap compaction, into also using an additional incremental and parallel compaction method. The purpose of this is to decrease the maximum pause times without too much loss of performance, since they claim that compaction is a major, possibly dominant, contributor to garbage collection pause times. The following can be said about their method:

1. The article extends an existing mark-sweep method for IBM's JVM from supporting only full-heap compaction, into also using their additional incremental and parallel compaction method
2. The existing mark-sweep method used the mostly concurrent method from [Boeh91], which seems related to [Boeh88] and the Boehm-Demers-Weiser sweeper mentioned earlier in section 4.5 [1]
3. The existing method also used compaction avoidance techniques to reduce compaction
4. Compaction in the existing method was done by a sequential method (by Morris), which is the method that they extend by also adding their parallel incremental compaction method. It does not replace full compaction, but hopefully replaces most of its invocations
5. Their is used method with a non-generational garbage collection algorithm, but is applicable to mature area garbage collection as well
6. Their incremental compaction reduces maximum pause times in three ways:
   (a) They only compact part of the heap at a time
   (b) All compaction phases run in parallel on all processors
   (c) If mostly concurrent marking is active, then collection of data on behalf of compaction is done concurrently with the mutators
7. The area to be evacuated for compaction is chosen before the start of the mark-phase and a fairly large data structure for Evacuated References (ER) is kept during the garbage collection cycle

For small heap sizes, they do not get any improvements in maximum pause times and they even get a slight performance hit. When the heaps get slightly larger ("reasonably large"), the maximum pause times are reduced without much degrade in performance, compared to when using only full non-incremental compaction. However, their best pause times, for some of the benchmarks, are still above

[1] I have not read up on these methods, so I could possibly be mistaken here

5ms and their pause times are also seen, for other benchmarks, to be around 100ms or more. Hence, this method does not seem to be sufficient in itself for interactive or real-time purposes.

6.9 Mark-Region and Immix

The article [Blac08], which I would venture to classify as a seminal paper, introduces a new class of memory management methods, which they call *mark-region*. It is similar to mark-sweep, except that memory is allocated in regions, rather than one single heap space. For this class of methods, they also introduce *opportunistic defragmentation*, which mixes copying (evacuation) and marking in a single pass. Their implemented method, which combines mark-region with opportunistic defragmentation, is called Immix. The key insight is to allocate memory at a coarse block (region) grain when possible and otherwise in groups of finer lines.

They evaluate the three canonical tracing collectors: semi-space copy collection, mark-sweep and mark-compact. Each of these collectors are claimed to sacrifice a performance goal: mark-compact has slow collection, semi-space copy collection is space-inefficient and mark-sweep has poor locality (and thus get slow mutator time), due to non-contiguous allocation. As we have seen in this thesis by now, however, these things may depend on which optimization methods are added to the methods, so what is referred to here are really the canonical methods, without added improvements. They show that their Immix collector, combining plain mark-region method with opportunistic defragmentation, is able to achieve all three goals sacrificed by each of the canonical methods. Thus they achieve space efficiency, fast collection and mutator performance, where the latter is due to contiguous allocation.

The basic mark-region method can be summarized as follows:

1. Divide memory into fixed-size regions, each of which is either free or unavailable
2. Bump allocate into regions until all free regions are exhausted, which triggers a collection
3. Regions containing at least one live object are marked as unavailable and all others marked as free
4. They claim that defragmentation is essential, since it reduces fragmentation overhead
5. The dilemma of region size is addressed by recycling partially used blocks (regions), by skipping over unavailable lines and allocating into contiguous free lines
6. Objects may span lines, but not blocks (regions)

The mark-region algorithm in Immix, which supports multi-threading, can be outlined as follows:

1. It consists of the following main parts or phases:
   (a) Initial allocation: Each thread allocates in one block at the time, until no more free (global) blocks are available
   (b) Identification: Clear all marks, then trace from all roots, marking objects and lines in object and line maps
   (c) Reclamation: A coarse-grained sweep identifies entirely free blocks, which are returned to the global pool of blocks. Blocks with free lines (i.e. partially free blocks) are *recycled*
   (d) Steady state allocation: Threads resume allocation into recycled blocks, by linear scan of the line map for holes

2. Allocation is correspondingly simple to a semi-space copying garbage collector, using a *bump pointer* and a *limit pointer*. The limit is either the next occupied line or the end of the block. These two pointers are updated when the limit is about to be crossed

3. Statistics from previous garbage collection cycle are used to select candidates for regions to defragment and for selecting evacuation targets. They call this lightweight *opportunistic evacuation*. When evacuating objects for defragmentation, new space for them is allocated in the same way as the mutator allocates. A few blocks (2.5% of the heap) are set aside as extra head room for evacuation. These blocks are therefore never returned as free

4. They select defragmentation candidates based on the number of holes in regions and other heuristics, since e.g. many holes indicate fragmentation

5. Marking and evacuation is done in the same pass. They state in section 3.2.1 in [Blac08] that, even without defragmentation, at least one mark bit is needed per object (not just per line), to ensure that the transitive closure terminate. As we shall see in the method implemented in this thesis, fewer mark bits can actually be enough in a language like Standard ML

6. When lines are marked in the line bitmap, they use conservative marking, by skipping over the first line in each hole at allocation time. This implicitly always marks one extra line, which could waste nearly a line per hole, but it significantly speeds up marking of small objects; this will be further explained in the method implemented for this thesis. They found that line marking is best done when scanning objects, rather than when marking them

7. The mutator may *pin* objects, which prevents them from being moved in memory. This is required by some programming languages, such as C#, while other languages might optimize on non-movable memory areas, e.g. for buffering in file I/O. Live objects are only evacuated if the mutator has not pinned it

8. Synchronization is required for parallel defragmentation (as with any kind of evacuating garbage collector), but they use trace specialization in MMTk, so this overhead is only paid for when actually performing defragmentation

9. A block is the unit of coarse-grain space sharing among threads, so each block acquisition and release must be synchronized and threads cannot share space at a finer granularity than a block

10. It is stated in section 3.1.1 in [Blac08] that the following is important for high performance: Recycling policy, allocation policy, demand-driven overflow allocation and parallelism (where the design maximizes the fast thread-local unsynchronized activity)

11. Blocks have a size of 32 kilobytes, but the method is not too sensitive to this. Lines are 128 bytes and the method is somewhat sensitive to this, which may depend more on the source language than architecture. Large objects of 8 or more kilobytes are handled specially, so Immix only considers objects less than 8 kilobytes. The worst-case block level fragmentation (due to block size and large object size) is 25%. The average worst-case internal fragmentation (line size versus block size) is 25%

They evaluate Immix as a full-heap garbage collector and two composite garbage collectors. Comprehensive benchmarks show that the full heap garbage collector consistently outperforms the best of the canonical collectors by 5-10% on total performance on average accross many heap sizes and that

it rarely degrades in performance. It requires on average only 3% more space than mark-compact and 15% less than mark-sweep. It matches the mutator locality of semi-space copy collection and matches the collection time of mark-sweep. They also build a generational garbage collector with an evacuating semi-space copying nursery and an Immix mature-space (which is not even heavily tuned), which performs slightly better on 20 benchmarks than if using mark-sweep for the mature-space (Jikes' best performing production garbage collector). It also significantly improves several interesting benchmarks. Their other composite garbage collector is an in-place Immix collector, which uses sticky mark bits, as described in [Demm90]. The sticky-mark-bits algorithm mostly performs partial garbage collection of the most recently allocated objects and only occasionally performs full heap collection. This in some sense makes it similar to a two-space generational collector, without having to introduce two different garbage collection methods and associated hybrid write-barriers, which are typically slightly more expensive than non-hybrid write-barriers, as was shown and argued earlier in section 4.11.4. Their only difference with [Demm90] is that they use an efficient object-remembering barrier, as described in [Blac04b] (and shown earlier in section 4.11.4), to identify modified mature objects, rather than page protection and card marking. Their in-place generational Immix collector almost uniformly improves over the full heap collector. The generational in-place Immix collector also performs very competively to their generational collector with an evacuating semi-space copying nursery.

An important performance aspect, which is not evaluated in the article [Blac08], is the duration of garbage collection pauses. Keeping pauses short is very relevant for use with interactive applications. I would however like to state that the evaluation in the article [Blac08] is otherwise the most thorough evaluation I have ever seen in a research article, even in different research fields, so it deserves credit for that. To get a taste of their thoroughness, they evaluate on three different hardware platforms (Intel Core 2 Duo, AMD Athlon and IBM PowerPC), whose properties are fully specified, down to the kind and frequency of the memory hardware and all cache and cache-line sizes. All platforms run the same operating system, Linux Ubuntu 7.04. Their experimental setup is carefully optimized, with stand-alone machines with all unnecessary daemons stopped and the network interface down. Their experiments were run six times. They discard the fastest and the slowest experiments and report the mean of the remaining four experiments. They display the 99% confidence intervals on their graphs. They also took several steps to control the compiler's dynamic lazy compilation in the Jikes RVM system, which is relevant for such a compiler. They compared their compilation configuration with that of Sun's 1.5 and 1.6 production JVMs, where their performance was in-between those two (5% faster than JDK 1.5, but JDK 1.6 is 12% faster than theirs).

In comparison to other work, they state in section 2 in [Blac08] that the Ulterior Reference Counting method from [Blac03b], which will be presented later in section 6.10.1, achieves pause-time guarantees, space guarantees and collector efficiency, but not locality, due to its use of segregated free lists. The lack of locality will in fact apply to many proposed methods in this thesis, since many of them either rely on segregated free lists or might use them as an optimization of other parts of the algorithm. The defragmentation of the Metronome system from [Baco03], which was presented earlier in section 6.7, is similar to the one in Immix, but requires exact fragmentation guarantees from a previous pass, whereas Immix is more dynamic in this respect (stated in section 2 in [Blac08]).

In summary, the Immix collector performs very well and is remarkably consistent and robust in its performance, where only its pause times are left unevaluated.

Very recently, an article about a garbage collector named Schism [Pizl10] was published, which is based on the mark-region and Immix method. This method is concurrent and aims for hard real-time performance, where high priority threads must be able to preempt everything in the system, including the garbage collector. They achieve short pause times, e.g. less than 10ms in the worst case for the

version of their system with the longest pause times, so it is not an inherent limitation of the mark-region method to achieve short pauses. The article [Pizl10] has a very thorough evaluation, which they believe to be the most thorough ever for real-time garbage collectors. Since the method implemented in this thesis will be for a single-threaded compiler without any kind of concurrency, the Schism method will not be investigated any further.

6.10 Sticky Mark Bits and a Theory of Garbage Collection

As was mentioned in the previous section, the technique known as sticky mark bits can be used for constructing in-place generational garbage collectors. The article [Demm90] explains this technique, but has several other contributions as well.

Generational collection is usually expressed in terms of copying objects. *Conservative pointer-finding* (as introduced earlier in section 4.1.1) usually precludes copying, because it conservatively classifies objects as pointers, even though they might be integers. The article combines these two techniques and develops both a framework for understanding garbage collection in general as well as two new garbage collection algorithms. Only tracing collectors are considered in this article.

In forming the basis for the theory in this article, they first define a storage model, consisting of an *allocated set* and a "*points-to relation*", as well as a few invariants. They define garbage collection, which can be either *precise* or *partial*. They also define *pointer augmentation* and have a theorem: every valid partial collection is equivalent to a precise collection on a pointer augmentation of the true storage state.

Posets (pointed partially ordered sets) and *embeddings* are defined and, using pointer-augmentation, they show how to construct *partial collections* from embeddings. In particular, it is shown how to: 1) collect only a fraction of the unreachable objects at a time, rather than a full collection and 2) perform generational collection, partially classified by *bystanders*, *immune objects* and *threatened objects*.

It is also described how to combine collection strategies by Cartesian products on embeddings. In this way, it is possible to e.g. make a collection strategy as simple and easy to compute as desired, while being as precise (with respect to pointer information) as some other strategy.

Turning to practice (in section III of [Demm90], they use their theories to define two new garbage collectors. For their partial ordered sets for both these collectors, they use the total order of allocation time, since this is natural for defining generational garbage collectors. Both of their implementations use a trick of temporal causality of pointers: newer objects can only point to older objects, unless by an update of older objects, which they can keep track of by virtual memory page protection or dirty bits. Both implementations also use "cards" and per-card information, rather than per-object information. In some sense, the lines used in the mark-region method Immix from the previous section can be seen as an instance of cards. This can give imprecise information and they call this precision loss "card pollution". Each of their collectors avoid card pollution in a different way.

Only the first garbage collector, presented in section IV in [Demm90], will be briefly described here, since this is the collector which uses the sticky-mark-bits method. This collector is an existing mark-sweep collector from a previous article, which they turn into a generational collector. The simple way in which this is done is by sometimes simply neglecting to clear the mark bits when performing a new collection, such that only a partial collection of the objects allocated since the last collection is made. This requires that they remember which objects were modified since the last collection. This could be done by a remembered set, but they use virtual memory page protection, where their "cards" are the memory pages. Instead of clearing the marks before the collection, they then mark all objects reachable from the objects, which were modified since the last collection. In their implementation, this means

that they mark all objects reachable from all memory pages containing objects, which were modified since the last collection.

The policy that they use for deciding between when to do a full heap collection and when to do a partial collection is the following: a partial heap collection is performed every time approximately 100 kilobytes of memory has been allocated since the last collection and a full heap collection is performed when all their cards are at least 3/4 (i.e. 75%) full, i.e. when they are running short of memory.

The sticky-mark-bits method is a general technique, applicable to several garbage collection methods. As argued earlier in section 4.11.6, using virtual memory page protection has been shown to generally be inefficient compared to using write-barriers in software, but fortunately the method is applicable also to garbage collectors using write-barriers. According to [Blac08], implementing the sticky-mark-bits algorithm for a mark-sweep collector did, however, not give as good results as when using it with their new mark-region method, so although the method is applicable to several garbage-collection methods, it may not be efficient for all methods.

6.10.1 Ulterior Reference Counting: Fast Garbage Collection without a Long Wait

The article [Blac03b] presents a hybrid garbage collector, combining incremental reference counting for long-lived objects (to achieve short pause times) with a copying generational garbage collector for short-lived objects (to achieve high throughput). Their reference counting method is an extension of deferred reference counting, which they call Ulterior Reference Counting (URC). Some information about their method is given here:

1. Their method is implemented in Jikes RVM with JMTk [Blac03a]
2. Their young generation is called a *nursery* and is a bounded (B) copying generational (G) collector, so it may potentially consist of several generations. The boundedness in the copying collector refers to bounded size, as opposed to flexible or fixed size, which they describe
3. Old memory objects is stored in a *mature space* and uses a variant of *deferred reference counting* (RC). Deferral is explained with the Zero Count Table (ZCT) versus the temporary increment approach. Buffering (which replaces ZCT) and coalescing are also described. Their deferred reference counting is described in terms of mutation events, retain events and deferral and it is integrated with the nursery through an integrate event. These methods are somewhat technical and all serve to reduce the number of reference counter increments and decrements
4. Bacon and Trajan's *trial deletion* from [Baco01] is used to detect cyclic garbage in their reference counting method. This has bounded time-usage, in order to avoid long pause times
5. They describe their method very generally, showing that the deferral policy may be chosen for each pointer and that the collection policy may be chosen for each memory object
6. A *mutated objects buffer* is used for recording each mutated non-nursery object (where non-nursery means either deferred or reference counted in this setting)
7. The nursery is collected with a Cheney-style breadth-first collector, which also traces from the mutated object buffer
8. The nursery is garbage collected when full, where survivors are integrated into mature space by enqueueing copied objects for scanning. Each scanned object triggers an integrate event

9. A write-barrier is used with fast (inlined) execution paths and slow paths, where only the slow paths require concurrency locks. Due to the way that this barrier interoperates with deferred reference counting, this is a somewhat specialized barrier, for which they show the code in figure 3 in section 3.2 in [Blac03b]. The fast versus slow paths in the barrier implementation is emphasized, as was also seen to be important earlier in section 4.11

10. Their write-barrier is not a garbage collection safe-point, so an asynchronous garbage collection trigger may be used

11. Their mature space allocator uses size class segregated free lists with size classes chosen such that the worst-case internal fragmentation is $\frac{1}{8}$ (i.e. 12,5%)

12. Objects larger than 8kb are separately allocated in a Large Object Space (LOS) and they allocate directly into this space for these objects

13. Their memory object headers are 2 bytes, with an additional 4 bytes for reference counters

Recall from earlier in section 3.3 that it is in this article that they state the generational hypothesis, that young objects mutate (i.e. are updated or created) frequently (with reference to [Appe89] and [Stef99]) and die at a high rate (with reference to [Leib83] and [Unga84]). They also state that old objects mutate infrequently and and die at a slower rate (with reference to [Appe89]), which they also observe, according to their measurements section 4.5 in [Blac03b]. These measurements are, as also stated earlier in section 3.3, for Java and the SPEC JVM benchmarks, so they could potentially be specific to imperative or object oriented programming languages or specific to that set of benchmarks.

Apart from presenting their new method, they also compare with a highly responsive deferred reference counting (RC) garbage collection algorithm and with pure mark-sweep (MS) and with a bounded (B) copying generational (G) mark-sweep (MS) garbage collector, BG-MS for short. They state that the latter is a popular choice for high performance garbage collectors. They find that their BG-RC collector has much better responsiveness (lower pause times) than BG-MS and that it matches and sometimes beats its throughput. BG-MS only gets better performance for small heaps. Reference counting is much slower, when also used for the nursery. The non-generational garbage collectors (RC and MS) have both worse thoughput (especially RC) and longer pause times (especially MS) than the generational hybrids (BG-RC and BG-MS). They also show that their mutator utilization is good.

It is stated that the only other generational reference counting algorithm that they are aware of is Azatchi and Petrank's algorithm [Azat03], which they claim give good pause times but lower throughput. They also compare with a couple of other previous works, where their method performs better. They believe that using a generalization of their Ulterior Reference Counting as the last belt in a Beltway configuration (for which they give a reference to the Jalapeño Virtual Machine) could perform even better than their current method.

In summary, they claim that their method is the first to achieve success in both low pause times and high throughput.

6.11 Code-Entry Barriers and Incremental Garbage Collection for Haskell

The article [Chea04] explores generational garbage collection for the functional programming language Haskell, which has lazy evaluation, as opposed to the strict evaluation of Standard ML. They build and

improve on their earlier work from [Chea00] and incorporate their methods into the Glasgow Haskell Compiler (GHC) [GHC].

Their basic method for achieving performance and low pause times is incremental garbage collection in the style of Baker's algorithm. The earlier method from [Chea00] exploits the dynamic dispatch mechanism of the GCH-compiler [GHC] to avoid the read-barrier by a kind of "self-scavenging code".

The dynamic dispatch mechanism referred to here is the use of *suspended computations*, also known as *thunks*, which is normally used in relation to lazy evaluation. In Standard ML, a thunk can be implemented explicitly by prefixing code with `fn () => ...`, i.e. the equivalent of putting a λ-abstraction in front of the code, in reference to the λ-Calculus, which we shall not delve into here. Lazy evaluation uses thunks everywhere in its computation as an inherent part of the language's semantics, in order to delay all computations until the time that they are actually needed, *if* they turn out to be needed at all. This method of optimization is therefore fairly specific and mostly beneficial for languages with a lazy evaluation semantics.

In making their earlier method, they had to introduce a few tricks to make it work, which gave some overhead. The article [Chea04] introduces specialized code generation, in order to eliminate some of this overhead (in space and time), which comes at a cost of an increase in the size of the static program code (code bloat). Their new method interacts with the methods used in the GHC compiler in various subtle ways. In section 5 of [Chea04], they propose using specialization to eliminate write-barriers for generational garbage collectors. This, however, is not worthwhile in their case, but it might be for write-intensive programs. They argue in section 4.5 that their performance gain in avoiding the write-barrier is only around 1.7%.

By removing the write-barrier, they want to achieve two effects: 1) there should be no write-barrier when updating thunks in the young generation and 2) when updating an object in the old generation, the thunk should be added to the remembered set automatically. They achieve this by specializing thunks into four different versions, which is what gives the aforementioned code bloat. They reduce the code bloat by sharing the code which is common to each of the versions, which gives almost identical performance.

For their method, they observe that execution times can sometimes be *reduced* by a substantial factor for their incremental garbage collector, compared to a stop-and-copy garbage collector, which they believe is the first time to have been reported.

One of their main results is that a 25% code bloat over stop-and-copy (15% over their previous method) gives an incremental garbage collector, which is 4.5% slower than a stop-and-copy collector and 3.5% faster than their previous method. They can achieve hard pause bounds, but handling large objects still requires some challenges to be met.

As can be seen from these descriptions, the optimizations are mostly related to when using thunks and lazy evaluation. The methods are therefore not particularly suited for implementing a memory-management system for a language with strict evaluation, such as Standard ML and CeXL, which are the languages considered in this thesis.

6.11.1 Others

The methods in the articles such as [Chen01], [Fram07] and [Spoo05] might also be relevant to consider. I have, however, not had time to delve into these methods.

Chapter 7

Design Analysis

This section gives a short overall design analysis, with the purpose of identifying suitable methods for implementation for this thesis.

7.1 Ruling out Entirely Static-Analysis-Based Methods

Section 5 had a somewhat detailed treatment of some possible ways of handling memory management by static program analysis. The main conclusion regarding those methods is that they do generally not form comprehensive solutions to the problem of performing memory management. Generally speaking, only when they are combined with some form of dynamic memory-management method, normally garbage collection, do they form a comprehensive solution. Hence, when choosing methods for implementation, we will mostly just be concerned with finding an appropriate garbage collection method. Types and static program analysis are only used and suggested as improvements to the garbage collection method. One significant such improvement, which has been implemented, is performing statically computed tracing of the root set for the heap data. This eliminates certain checks at runtime, which may hopefully give a small increase in the speed of root-set tracing. Statically computed tracing also makes it fairly natural to facilitate mostly tag-free garbage collection. The price paid for this is extra specialized code generated at compile-time, which increases the size of the final program somewhat. Having implemented this feature meets goal number 5 from section 1.3.

7.2 A Few General Considerations and Engineering Efforts

Methods like Nettles and O'Toole's system from section 6.3 and the Huelsbergen and Larus collector from section 6.4 seem to rely on generational garbage collection (with more than two generations), in order to get their good performance. Building a general (more than two-generation) generational garbage collector seems to take a larger engineering effort than if this can be avoided, so although it could give improvements, such a many-generation generational garbage collector will not be implemented.

Methods like Metronome from section 6.7 and Ulterior Reference Counting from section 6.10.1 seem more convincing, since they get their performance by desiging the system to give the required performance all the way. Hence, aiming for methods like Metronome and Ulterior Reference counting seem like better choices than the other systems mentioned here. However, these two methods also represent a large amount of work, since they consist of many separate methods and optimizations. It therefore also might be unfeasible to implement such methods, at least in their entirety. Another consideration is that these systems have already been constructed and evaluated by others, so from a

scientific point of view, it would be more interesting to try to create something which at least to some extent differs from previous work.

As a final consideration on engineering efforts, support for concurrency, multi-threading or symmetric multi processing will not be implemented, as was also already stated in section 1.3 for goal number 4.

7.3 Nursery and Mature Generation Hybrid Designs

If the main design is to use a hybrid garbage collector, based on two generations, one for young (nursery) short-lived memory objects and one for mature long-lived memory objects, I would suggest making the first generation some kind of a copy collector, which has been successfully used in other methods, e.g. Ulterior Reference Counting from section 6.10.1 or the Doligez-Leroy-Gonthier collectors from section 6.5. For the mature generation, I would recommend one of the following:

1. A mostly non-moving mark-sweep collector, similar to what gave good results in the article [Baco03]

2. Deferred Reference Counting, which gave even better results in the article [Baco03] than using mark-sweep, as suggested above. This forms the Ulterior Reference Counting method

3. Baker's Treadmill. This method suffers from fragmentation and lack of ability to handle variable-sized objects, so it would likely have to be combined with some kind of occasional compacting (to handle fragmentation) and multiple treadmill instances for segregating into object size classes. Even just segregating into object size classes actually makes each threadmill have well-defined "slots" of memory spaces, meaning that fragmentation should not even be a problem for allocation. Thus, defragmentation is mostly needed for reclaiming whole pages of memory to the external heap manager, when memory residency (occupancy) is low. For handling this, the buddy system, mentioned in section 4.4.3, might be useful

Implementing any one of these suggested methods for the mature memory objects would likely even be adequate for interactive performance in themselves. Thus, adding a copy collector for young memory objects could be seen as an optimization, for achieving higher performance.

The sticky mark bits algorithm from [Demm90] is another way of turning a garbage collector into an in-place two-generation collector, which could therefore be seen as another way of achieving an optimization for higher performance. However, as mentioned previously in section 6.10, it may not give good performance for all methods, since e.g. it did not give particularly good performance for a mark-sweep collector evaluated in [Blac08].

7.4 The Immix-Style Mark-Region Method Achieves Everything

The mark-region method and Immix, which were quite extensively described in section 6.9, seems to achieve many of the desired goals. Looking at the goals in section 1.3, the following can be concluded:

1. It is a general purpose memory-management method for a compiler (goal 1) making no assumptions about the application being compiled (goal 11)

2. There are also no issues involved in using this method for a functional programming language (goal 10). The memory barriers that it uses are also write-barriers, not read-barriers, which seems like the best choice for a functional language

3. According to [Blac08], it has high performance (throughput) (goal 3), low space-overhead (goal 6) and good locality of reference, which gives good cache and paging behaviour (goal 7)

4. The implementation in [Blac08] supports multi-threading with efficient synchronization, so although multi-threading will not be implemented for this project, the method scales to this, which meets goal 4. Also, the article [Pizl10] confirms that the mark-region method even scales to concurrent multi-threaded real-time memory management

5. It also seems that there would be no problem in supporting a binary interface with this method, which makes it possible to create dynamic link libraries (DLLs) (goal 9)

6. The article [Blac08] does not evaluate pause times (goal 2), which is very important for this thesis. However, [Pizl10] achieves hard real-time constraints with a derived method and there does not seem to be anything inherently preventing the method from achieving good (i.e. short) pause times. The in-place generational garbage collection using sticky mark bits from [Demm90], as also analysed in [Blac08], will be suggested for improving the performance and for making longer pause times rarer. Incremental stack tracing will be proposed for limiting the pause times of the major collections in the implementation. Since the article [Blac08] did not evaluate pause times for the method, evaluating pause times will give a relevant scientific contribution for this thesis

If the pause times end up being sufficiently short in the implemented method, this should hereby achieve all desired goals set for this thesis, which is actually quite remarkable, given the nature of the memory-management problem, where many trade-offs and compromises usually have to be made.

7.5 The Choice of Method

For the reasons presented in the previous section, the Immix-style mark-region method seems like an ideal candidate to choose for implementation, which is precisely why it is the chosen method. Some of the other designs listed in section 7.3 would also be relevant and could likely give adequate performance for interactive settings, some of them maybe even better with respect to pause-time guarantees. However, those methods would also likely constitute a larger implementation effort and it is certainly not given that they would perform any better with respect to other performance parameters, especially given the very promising results from [Blac08] and, for hard real-time systems, from [Pizl10].

Chapter 8

The Implemented Methods

This section presents the methods implemented in this thesis.

8.1 Introduction to the Typed Intermediate Languages *FILM*$_H$ and *FILM*$_L$

In the compiler implemented for this project, there are two important typed intermediate languages, which are used for the middle-end of the compiler. These are called the *FILM*-languages, for Typed **F**irst-order **I**ntermediate **L**anguage for the **M**iddle-end of the compiler. The *FILM* languages come in two flavours: A high-level version, *FILM*$_H$, and a low-level version, *FILM*$_L$.

In this thesis, part of the memory management system is generated as *FILM*$_L$-code, which is the intermediate language used just prior to the assembly code generation. Since pseudo-code is shown in the *FILM*$_L$ language for part of this generated code, this language will be briefly introduced here.

The syntax for the low-level *FILM*$_L$ language is presented in figure 8.1. In this language, $scon$ may be integer values, word values, char values, bool values, real values (which are however currently not supported in the back-end) and string pointers. The patterns p in the **case**-expression may have $scon$ values of integer, word, char, bool and string pointers, but not reals. Patterns of the form $w(x : \tau)$ are currently not used in the *FILM*$_L$ language. The syntax of the type system is defined as follows:

τ	$::=$	$styp(\tau_1, \ldots, \tau_n) \mid [\tau]_m \mid [\tau]_c \mid$	Types: Built-in ($n \geq 0$), mutable pointer, constant ptr.,
		$\{w_1 : \tau_1, \ldots, w_n : \tau_n\} \mid$	product types ($n \geq 0$),
		${}^{+}d\{w_1 \langle : \tau_1 \rangle + \ldots + w_n \langle : \tau_n \rangle\}$	named tagged sum types ($n \geq 0$)

Notice that function types are not defined as part of the type system, since types are separately specified for function parameters and return types. This makes it a first-order language. The built-in types $styp$ are at least exn, integer, word, char, bool, real and string, which are all unparameterized, i.e. $n = 0$. $styp$ may also contain other primitive types, such as arrays. The array type (and similarly the type `ref`) has one type parameter, the type of elements in the array, so $n = 1$ in this case. Pointer types (boxing of data) are explicitly specified and may be either pointers to mutable (modifiable) data or pointers to constant data. The product types are described by a set of typed fields, each specified with a memory offset of type word, w_i, from the start of the product type described. The tagged sum types are described by a name, d, and a set of tag values of type word, each one optionally specified with the type of value that it stores, when the runtime value contains that tag. The optional types for the tags is because these cannot necessarily be inferred directly from ξ-Calculus, so they are not always

a	::=	$x \mid scon$	Atomic expression: Variable, constant
e	::=	**let** fs **in** e | **let** $x : \tau = r$ **in** e | $M[a+i] := a_v$; e | **return** $x(a_1, \ldots, a_n)$ | **return** a | **raise** a | **if** a_1 $relop$ a_2 **then** e_1 **else** e_2 | **case** x **of** $p_1 => e_1 \parallel \cdots \parallel p_m => e_m$ $\langle \parallel \_ => e_{err} \rangle$	Expression: Function declarations, typed declaration, memory store with offset, tail-call ($n \geq 0$) (currently unsupported), return, raise exception, conditional branch expression, case expression ($m \geq 1$) with optional default clause
p	::=	$scon \mid w(x : \tau)$	Patterns: Constant, parameterized word const.
r	::=	$a \mid M[a+i] \mid w(a) \mid$ $x(a_1, \ldots, a_n) \langle$ **handle** $x => e_h \rangle \mid$ $prim(a_1, \ldots, a_n) \langle$ **handle** $x => e_h \rangle$	Returning exp.: Copy, memory access, constructor, calls to declared functions ($n \geq 0$), primitive function calls ($n \geq 0$) (non-tail calls have optional exception handlers)
fs	::=	**fun** $x(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = e$ $\langle fs \rangle$	Mutually recursive typed function declarations (a sequence of delcarations)
$prim$	::=	`___Word.+`, `___print`, `___MemAlloc`, ...	
$relop$	::=	`___Word.<`, `___Word.=`, ...	

Figure 8.1: The syntax for the low-level $FILM_L$-language, which is the typed intermediate language in the compiler used immediately before generating assembly code interacting with the memory management system

specified. Also, constructor types can never be fully specified for recursive types, since this would yield an infinitely large type, so the type syntax must allow this to be unspecified, hence the optionality.

The language *FILM*$_H$ is quite similar to *FILM*$_L$, except that it has a more high-level view of records and constructors. The memory layouts of records and constructors are not specified in *FILM*$_H$, except by label and tag names, and the exact memory operations for manipulating them are not specified either.

I have a couple of planned changes and enhancements for these two languages, where one is related to letting functions return tuples instead of just a single value, just as they take a tuple of parameters. Another planned change is related to letting **if**-expressions and **case**-expressions return values, such that inlining functions with these expressions becomes more feasible, but I am less certain of whenther that would be a good idea or not than I am with the first suggested change. Further descriptions of this will be saved for the upcoming report [Anoq10b]. For this thesis, the languages are as set in stone.

When *FILM*$_L$ pseudo-code is shown in this thesis, it will not always be typeset in the mathematical style shown here, but rather as verbatim ASCII-text. This should hopefully not cause confusion, since the language syntax is partly designed to be ASCII-printable.

8.2 An Overview of the Implemented Compiler

The following is an overview of the phases of the implemented compiler:

- The compiler front-end, which parses, type-checks and transforms CeXL programs into ξ-Calculus, as specified by the Design and Definition of CeXL and ξ-Calculus version 0.9.3 [CeXL10]

- The ξ-Calculus compilation phases, which roughly are: Monomorphization, pattern-matching compilation, closure-flow analysis and closure conversion. The closure conversion transforms programs into the *FILM*$_H$ typed intermediate language. The monomorphization phase also performs dead code elimination, which is good enough to eliminate an entire 1,000-line basis library, if it is not used by the compiled program

- The mandatory initial transformations in *FILM*$_H$ are: 1) transforming tagged sum type booleans and **case**-expressions into **if**-expressions and primitive booleans and 2) flattening the record arguments passed to built-in functions, such that the record fields are passed as a list of arguments. The *FILM*$_H$ program may be type-checked after these phases, in-between all the following phases. The type-checker has been very useful for catching many mistakes in the compilation phases and, somewhat to my surprise, the mistakes are usually not in propagating the types correctly (it is apparently actually somewhat difficult to propagate types wrongly), but mostly arising out of true errors in the transformations

- A *FILM*$_H$ transformation, which eliminates exceptions, by wrapping all function return types into two-constructor summed tag types, with one constructor for return-paths and one for raised exceptions, and inserting **case**-expressions after all non-primitive function-call-sites. This assumes that no primitive operations will ever raise exceptions, which is reasonable to assume for the simple benchmark programs that we will consider. This transformation avoids having to implement exception support in the compiler back-end and thus also for the root set tracing, which simplifies the construction of the memory-management system

- A simple *FILM*$_H$ optimizer, which iterates dead code elimination with code specialization, where the latter consists of copy and constant propagation and specializing inlining. The iteration continues until no more inlining occurs. This is a minimal a set of optimizations, which was quite

necessary, in order to be able to understand and debug the generated code, since in particular the closure conversion and translation from ξ-Calculus to $FILM_H$ generates lots of small functions and coercions, which makes the intermediate code very hard to read and understand. Many of the coercions are optimized away, but much of the code resulting from the exception elimination is still not optimized away and still clutters the code quite a lot

- Translation from $FILM_H$ to $FILM_L$, by freezing the runtime memory representation of all data. $FILM_L$ programs after this phase may still be type-checked, but the type-check does not guarantee that the generated program does not access invalid memory addresses
- A $FILM_L$ phase for eliminating **case**-expressions with only a single branch. This is the first time in the transformation phase, where eliminating such expressions becomes possible, since the $FILM_H$ language can only deconstruct tagged sum types by **case**-expressions. $FILM_L$ directly reads the memory addresses of constructor-carried values, which can be done without **case**-expressions
- A $FILM_L$ pass to simplify **case**-expressions into **if**-expressions, in order not to have to implement jump-tables in the back-end, which would be needed for supporting **case**-expressions efficiently
- A simple memory-management specific pass from type-checked $FILM_L$ code, which is performed just before assembly code generation, as described later in section 8.5
- Assembly abstract syntax code generation. Assembly code is currently generated for the 32-bit Intel x86 processors, i.e. using the IA32 instruction set. In this phase, assembly instructions are directly generated for some primitive operations, such as the arithmetic and comparison operators, while other primitive operations are handled by generating calls to library routines written in C
- The x86 assembly code is generated directly by exporting the x86 abstract syntax into GAS (GNU Assmbly) format, suitable for use with the C-compiler `gcc` and the assembler `gas` on Linux. A small debugging tool was developed here, which simultaneously with the assembly code generation gathers a simple stack layout from the generated instructions. The stack layout contains information for each function about which assembler code line writes to a each stack frame offset in the function. This has been handy for debugging

The following features are currently not supported in the compiler:

- Reals, since this requires somewhat special handling for x86 code generation
- Exceptions are not supported in the back-end, but exceptions written by the programmer in the program or generated by the compilation, e.g. for pattern-matching compilation, are eliminated and will thus work
- References are not supported, although this should hopefully only take a bit of debugging and minor changes to get to work
- Arrays are not implemented. It would take a bit of debugging and also implementation of a few array primitives, such as `Array.sub`, `Array.create` and `Array.update`, to support arrays

The only places where write-barriers would be needed in the memory-management system is for references and arrays, so it is quite relevant that at least one of these is supported for a realistic evaluation. The other limitations are partly due to the lack of time to complete the compiler back-end and partly serve to simplify construction of the memory-management system. Simplifying the memory-management system is considered a good thing, given the limited time-frame of this project and the large amount of work that was needed just to get the compiler up and running until this stage.

8.2.1 Details on Handling Closure Records in Recursive Calls

There are many ways of handling closures in functional languages. Many of the ways of handling closures for recursive calls result in cyclic data structures in the heap at runtime, as was also mentioned earlier in section 5.4. This is actually not the case for the compiler implemented for this project. Since handling closures for recursive calls was not properly described in the report [Anoq10a] and since the planned revised version of that report [Anoq10b] is not yet available, the method will be described here.

Closures for a single set of mutually recursive functions are handled in the following way:

1. One closure record is created with all free variables for the entire set of mutually recursive functions. This closure record thus holds adequate closure data for any one of the recursive functions

2. The closure record for the set of mutually recursive functions is wrapped with a different data constructor and bound to a different variable for each of the recursive functions

3. When functions are called, not just recursive functions, their closure value is dispatched by matching out relevant constructors, thus unwrapping its contained closure record. The called functions are given the unwrapped closure record as an extra parameter

4. At the entry point of each of the recursive functions in the set of mutually recursive functions, the closure record is wrapped into a set of constructors bound to variables, one for each function in the mutually recursive set, exactly as it was done previously where the functions were declared. This allows recursive calls to be made within those functions, without any recursive data structures

8.2.2 Quantified Compilation Process for a Simple Example Program

As an example measure of the amounts of code introduced by the compiler, the line counts of intermediate and final code is given here after a few select compilation phases for a single specific example program, `fib35.sml`, which recursively computes the 35th Fibonacci number. The code-line counts are given in figure 8.2.

Program Stage	Language	Lines of Code (LOC)
fib35.sml source program	Standard ML	(excl. approx. 1,000 LOC basis) 14
Closure-converted program	$FILM_H$	442
Exception-eliminated program	$FILM_H$	1,170
Optimized program	$FILM_H$	209
Compiled optimized program	x86 assembly	4,692
Comp. opt. w/o stack-tracers	x86 assembly	854
Compiled unoptimized program	x86 assembly	21,958

Figure 8.2: Line counts of intermediate and final code after a few select compilation phases for the single specific example program *fib35.sml*

What can be concluded from these numbers is clearly that some of the early ξ-Calculus-based compilation phases generate a lot of intermediate code. Much of it stems from the coercions introduced by closure conversion and the functions introduced when generating the code for the first $FILM_H$ phase. The generated functions are required since the $FILM_H$ language is not able to continue execution of code after it has branched into either a **case**-expression or an **if**-expression, since these expressions do

not return a value and since the language cannot join multiple expression branches, as done e.g. by ϕ-functions in SSA-transformations (see section 19.1 in [Appe98]). The intermediate language presented in section 19.7 in [Appe98], which the *FILM*$_H$ language is greatly inspired from, has the same property.

It can also clearly be concluded by these numbers that eliminating exceptions introduces a fairly large amount of code. Fortunately, it can also be concluded that the optimization is able to get rid of significant amounts of code, more than a factor of five in this case. When inspecting the optimized code, it is also quite evident that it would not take much more work to enable the optimizer to reduce it significantly more. The compiled optimized program, which is used in the experiments, is included with the thesis but not shown in the appendix, since it would take up 60 pages with the smallest fontsize. The source program `fib35.sml` is included in appendix 12.1.

8.3 Direct Memory Access and User-Written Garbage Collectors

It is quite easy to extend the compiler to support direct memory access. As we shall see in this section, this can even be useful when implementing the memory-management system.

The primary features of direct memory access are explicit memory allocation and deallocation and copying of memory values to and from explicitly given memory addresses. A proposal for the signatures of two Standard ML modules for handling this is shown in figures 8.3 and 8.4.

```
signature UNSAFE_MEMORY_ALLOC =
sig
  val alloc : word -> addr (* Allocate a memory area of the given size *)
  val free  : addr -> unit (* Free memory area at the given address *)
end
```

Figure 8.3: A Standard ML signature for a module implementing explicit memory allocation. This is an unsafe module and the proposed structure name for this is *UnsafeMemoryAlloc*

The primary reason for splitting this into two modules is that additional modules might be desirable, e.g. to access the memory addresses of memory areas allocated in other ways, such as image buffer objects in graphics libraries, like the SDL (Simple Direct Media) Library. Another reason is that it is possible to write a somewhat safer version of the signature from figure 8.4, which is given in section 8.5. The proposal in figure 8.5 is, however, still not entirely safe, due to the possibility of specifying offsets to the functions and in particular due to the types of the functions `addradd`, `addrsub`, `addrtoword` and `wordtoaddr`. As an example, the functions `addradd` and `addrsub` may be used to change the type of memory value pointed to by an address, by calling them with an offset of zero, in which case a compiler should even easily be able to optimize the calls away. The calls `addrtoword` and `wordtoaddr` can probably always be optimized away, but are needed to be able to write garbage collectors masking out or marking bits in pointers. An entirely safe interface may give some limitations, since it may somewhat limit the programmer to only do well-typed operations to memory, which may not always be flexible enough. Also notice that the interface in figure 8.5 introduces a parameterized address type, which may not be directly compatible with that of the interface in figure 8.3. The allocation itself would also be a point of unsafety in these interfaces. In the rest of this thesis, only the module in figure 8.4 will be used.

One reason why one might want to have such modules is to allow writing programs of a more low-level nature with direct memory buffer access. One example is writing an interactive 3D renderer in Standard ML. I have written such a renderer, using the MLton compiler [MLton], where writing pixels

```
signature UNSAFE_MEMORY_ACCESS =
sig
  type addr          (* Opaque address type, only accessible through this interface *)
  type offset  = int (* Offset for memory addresses *)

  val addradd : addr * offset -> addr    (* Add an offset to an address *)
  val addrsub : addr * offset -> addr    (* Subtract an offset from an address *)

  val addrtoword : addr -> WordAddr.word (* Type-casts: WordAddr should be declared
  val wordtoaddr : WordAddr.word -> addr    as either Word32 or Word64 (struct assign) *)

  val read8     : addr * offset -> Word8.word  (* Read an 8-bit value from memory *)
  val read16    : addr * offset -> Word16.word
  val read32    : addr * offset -> Word32.word
  val read64    : addr * offset -> Word64.word
  val readaddr  : addr * offset -> addr        (* Read an address from memory *)

  val write8    : addr * offset * Word8.word  -> unit (* Write 8-bit value to memory *)
  val write16   : addr * offset * Word16.word -> unit
  val write32   : addr * offset * Word32.word -> unit
  val write64   : addr * offset * Word64.word ->
  val writeaadr : addr * offset * addr -> unit        (* Write address to memory *)
end
```

Figure 8.4: A Standard ML signature for a module implementing direct memory access. The primary features of this are copying of memory values to and from explicitly given memory addresses. This is an unsafe version of the signature and the proposed structure name for this is *UnsafeMemoryAccess*

```
signature SAFER_MEMORY_ACCESS =
sig
  type 'a addr       (* Opaque address type, only accessible through this interface *)
  type offset  = int (* Offset for memory addresses *)

  val addradd : 'a addr * offset -> 'b addr (* Add an offset to an address *)
  val addrsub : 'a addr * offset -> 'b addr (* Subtract an offset from an address *)

  val addrtoword : addr -> WordAddr.word (* Type-casts: WordAddr should be declared
  val wordtoaddr : WordAddr.word -> addr    as either Word32 or Word64 (struct assign) *)

  val read8     : Word8.word  addr * offset -> Word8.word  (* Read 8-bit value from mem. *)
  val read16    : Word16.word addr * offset -> Word16.word
  val read32    : Word32.word addr * offset -> Word32.word
  val read64    : Word64.word addr * offset -> Word64.word
  val readaddr  : 'a addr addr * offset -> 'a addr         (* Read address from memory *)

  val write8    : Word8.word  addr * offset * Word8.word  -> unit (* Write 8-bit to mem. *)
  val write16   : Word16.word addr * offset * Word16.word -> unit
  val write32   : Word32.word addr * offset * Word32.word -> unit
  val write64   : Word64.word addr * offset * Word64.word ->
  val writeaadr : 'a addr addr * offset * 'a addr -> unit         (* Write addr. to mem. *)
end
```

Figure 8.5: A Standard ML signature for a module implementing direct memory access. The primary features of this are copying of memory values to and from explicitly given memory addresses. This is a slightly safer version of the signature than the one proposed in figure 8.4 and the proposed structure name for this is *SaferMemoryAccess*. This module is still not safe, due to the possibility of specifying offsets and in particular due to the types of the functions *addradd*, *addrsub*, *addrtoword* and *wordtoaddr*

is done with a few custom-written C-functions that write directly into the frame buffer memory area. Using such custom-written functions, however, has the overhead of a function call. With the interface proposed here, on the other hand, such memory writes can be achieved by a single move instruction on most, if not all, processor architectures, assuming that the operands in question have already been loaded into registers and the primitive operations of the proposed interface are inlined. I suggested a more general version of "inline assembler code" on the MLton discussion list, which was responded to by Matthew Fluet on 2001-02-27 . The idea was rejected at that point, but it was admitted that peeking and poking memory would be easy support, which is the basic support given by the above interfaces. An alternative to the above interfaces would be to use the unsafe array modules supported by the MLton compiler [MLton]. However, this does not allow explicit deallocation of memory, which is relevant for the considerations further below. The unsafe array structures do also not allow updating the base address value of the array. Being able to update the memory address directly can give additional efficiency for some programs.

In the implemented compiler, at least a simplified version of these interfaces are intended to be written, consisting only of the functions `UnsafeMemoryAlloc.alloc`, `UnsafeMemoryAlloc.free`, `UnsafeMemoryAccess.addradd`, `UnsafeMemoryAccess.addrsub`, `UnsafeMemoryAccess.read32` and `UnsafeMemoryAccess.write32`. However, only the last four of these functions are used in the descriptions in the coming sections and only the two functions `UnsafeMemoryAccess.addradd` and `UnsafeMemoryAccess.addrsub` are currently implemented and actually used. The types `addr` and `offset` are not declared. One practical reason is that, the CeXL programming language currently neither supports signatures, nor opaque or abstract types, preventing the `addr` type to be opaque in the first place. Also, the type `word` is used instead of `addr` and `Word32.word`, since the compiler currently only supports `word`, which is a 32-bit value; it does not support e.g. `Word32.word`. It is very simple to incorporate this into the compiler. The allocation and free calls do not need to be very efficient and can be implemented as C-functions, while the memory reads and writes can be done as simply and efficiently as the primitive operations + and − or as memory loads and stores, depending on the operation.

An even more interesting reason for implementing support for these operations in the compiler is that it would allow a large part of the garbage collector, possibly at least allocation and the internal memory manager, to be written in the source language, Standard ML or CeXL, in this case. The read-barriers and write-barriers from section 4.11 were also defined in Standard ML using the modules presented here. To make it work in practice in the current compiler, it might be necessary to compile the relevant functions into $FILM_H$, dump them as abstract syntax, which is then included in the code during compilation, declared as functions with identifiable names, in order to know the name of the relevant functions. With a little more work, it might even be possible to let the programmer write part of the garbage collector, but this will not be a goal to achieve.

In the current implementation, the two implemented primitive functions are only used in the generation of functions for tracing the root set of the heap, which is described in the following section. The primitive operations are prefixed by `___` in the $FILM_L$ language, where these functions are used. There is also another primitive operation, called `___CallFunctionPtr`, which calls a pointer, given as the first argument, as a function, with the remaining arguments passed to that function. One final word of notice related to these primitive functions is that the function `___MemAlloc`, which will be used later, is not to be confused with `___UnsafeMemoryAlloc.alloc`, since the former is for the managed memory allocation, while the latter is for unmanaged explicit allocation, which requires explicit deallocation, i.e. something which the user (i.e. the mutator programmer) might wish to do in certain cases. The canonical example here is using `___UnsafeMemoryAlloc.alloc` and the other unsafe functions for implementing a garbage collector, which implements `___MemAlloc`.

8.4 Using Types to Avoid Tagging and Object Headers

The stack in the compiler grows downwards, from higher memory addresses towards lower addresses. The top illustration of figure 8.6 is a sketch of the stack-frame layout for the compiler when not implementing memory management or, in particular, root-set tracing.

To implement statically-typed root-set tracing, the stack frame is changed to what is illustrated at the bottom of figure 8.6, where the boldface entries are the new additions.

To achieve statically typed root set tracing, a function for tracing the root set of the current stack frame is generated at compile-time for each non-primitive function call-site in the program. A pointer to the tracing function is written to the stack frame just before the call of the call-site in question is performed. The generated function unconditionally traces all relevant addresses in the stack frame. When tracing of the entire stack frame is complete, it calls the tracing function stored in the previous stack frame, in order to ensure that the entire stack is traced. The program's main function (or the thread's main function, if supporting mutiple threads) is the only function which does not call any previous tracing function on the stack.

When performing stack tracing, each stack frame's tracing function can be implemented with the sketch shown in figure 8.7 as $FILM_L$ code, since the code is actually generated as $FILM_L$ code. The function `___CallFunctionPtr` is the special compiler-primitive function briefly introduced earlier in section 8.3, which calls the function-pointer given as the first argument as a function and has the remaining arguments passed to it. This implementation could use tail-call optimization, although the current assembly code-generation does not support tail-call optimization. The stack frames are traced top-to-bottom, i.e. upwards in stack addresses, since the stack grows downwards.

An alternative version of the sketch of a tracing function is shown in Figure 8.8. In this implementation, the actual stack frame tracing is done after the call to the next stack frame tracing function, which prevents tail-call optimization. On the other hand, with this bottom-to-top traversal order, the data in the heap is traced in the order in which it was created, which might give better cache and paging behaviour than the traversal order in figure 8.7. Hence, there is a trade-off to be made between which of these two functions should chosen. However, as we shall see later, only the latter traversal order, i.e. bottom-to-top, naturally supports making the tracing of major garbage collections incremental, which is important for keeping garbage collection pause times short.

The actual tracing of each stack frame was only shown as pseudo-code in figures 8.7 and 8.8, indicated by `<Trace the variables of the stack frame with base at stackPtr>`. This tracing for each stack frame is generated so as to unconditionally follow the entire memory layout structure of the type of each traced variable. There are only a few places where conditional tracing and tagging are needed:

- Tagged sum types may contain any of its sum-type tags in the runtime memory layout, so a conditional tracing function has to be generated, which traces each possible memory layout
- References may give cyclic data structures. Hence, a tag is needed to mark whether a reference has been visited during the runtime trace. Since references are pointers, the tag can be smuggled into the low order bit of the reference, without additional space requirements, assuming that allocated heap addresses are always aligned to at least 2 bytes. The generated tracing function becomes conditional at references due to this tag
- Arrays may both have dynamic length and give cyclic data structures. Just as for references, we need a tag to prevent infinite recursion. In the Standard ML Basis library, the function `Array.length` returns the length as a value of type `int`, which is signed, rather than the unsigned type `word`. Since arrays can never have negative length, we can therefore smuggle the tag bit

Previous stack frame	⋮	↑ Towards stack bottom (High memory addresses)
Stack parameters given to function A	Stack parameter PA_1 Stack parameter PA_2 ... Stack parameter PA_m	
Automatically placed by the x86 `call` instruction	Return address	
Function A's local stack frame	Local variable or spill A_1 Local variable or spill A_2 ... Local variable or spill A_n	
Stack parameters from function A passed to B	Stack parameter PB_1 Stack parameter PB_2 ... Stack parameter PB_m	
Automatically placed by the x86 `call` instruction	Return address	
Function B's local stack frame	Local variable or spill B_1 Local variable or spill B_2 ... Local variable or spill B_n	
Stack parameters from function B passed to C	Stack parameter C_1 Stack parameter C_2 ... Stack parameter C_m	
Automatically placed by the x86 `call` instruction	Return address	↓ Towards stack top (Low memory addresses)
Next stack frame for function C	⋮	

Previous stack frame	⋮	↑ Towards stack bottom (High memory addresses)
Stack parameters given to function A	Stack parameter PA_1 Stack parameter PA_2 ... Stack parameter PA_m **Given stack tracing function ptr.**	
Automatically placed by the x86 `call` instruction	Return address	
Function A's local stack frame	Local variable or spill A_1 Local variable or spill A_2 ... Local variable or spill A_n	
Stack parameters from function A passed to B	Stack parameter PB_1 Stack parameter PB_2 ... Stack parameter PB_m **Stack tracing function ptr. for A**	
Automatically placed by the x86 `call` instruction	Return address	
Function B's local stack frame	Local variable or spill B_1 Local variable or spill B_2 ... Local variable or spill B_n	
Stack parameters from function B passed to C	Stack parameter C_1 Stack parameter C_2 ... Stack parameter C_m **Stack tracing function ptr. for B**	
Automatically placed by the x86 `call` instruction	Return address	↓ Towards stack top (Low memory addresses)
Next stack frame for function C	⋮	

Figure 8.6: Illustration of the basic stack-frame layout of the call-stack. The illustration at the top is for when no memory management is supported. The bottom illustration is when pointers for typed root-set tracing have been installed, shown in boldface

```
fun topdown_frametracer(stackPtr : [word()]c) : int() =
    <Trace the variables of the stack frame with base at stackPtr>
    let nextFramePtr : [word()]c = ___UnsafeMemoryAccess.addradd(stackPtr, <stack frame size>) in
    let nextTracerFuncPtr : word() = M[nextFramePtr + 0] in
    let unused : int() = ___CallFunctionPtr(nextTracerFuncPtr, nextFramePtr) in
    return 0
```

Figure 8.7: Sketch of each stack frame's tracing function as $FILM_L$ code. Unspecified pseudo-code is enclosed in $\langle$ and $\rangle$, in this case the actual tracing of the stack frame's variables. The function *__CallFunctionPtr* is a special compiler-primitive function, which calls the function-pointer given as the first argument as a function and has the remaining arguments passed to it. In can be noticed that this implementation could use tail-call optimization, if the value *unused* was returned directly instead of the constant 0. The return value is ever used anyway, so this would be valid. The stack frames are traced top-to-bottom, i.e. upwards in stack addresses, since the stack grows downwards

```
fun bottomup_frametracer(stackPtr : [word()]c) : int() =
    let nextFramePtr : [word()]c = ___UnsafeMemoryAccess.addradd(stackPtr, <stack frame size>) in
    let nextTracerFuncPtr : word() = M[nextFramePtr + 0] in
    let unused : int() = ___CallFunctionPtr(nextTracerFuncPtr, nextFramePtr) in
    <Trace the variables of the stack frame with base at stackPtr>
    return 0
```

Figure 8.8: Sketch of a stack frame's tracing function as $FILM_L$ code, when the heap tracing order among stack frames is reversed compared to figure 8.7, such that the stack frames are traced bottom-to-top. Notice that the actual stack frame tracing, unspecified and enclosed in $\langle$ and $\rangle$, is done after the call to the next stack frame tracing function, which prevents tail-call optimization. On the other hand, the data in the heap is traced in the order in which it was created, which might give better cache and paging behaviour than the traversal order in figure 8.7

into the sign bit of the array length without additional space overhead. As was also the case for references, the generated tracing function thus becomes conditional at each array occurrence. Furthermore, due to the dynamic length of arrays, the array entries are traversed in a dynamically terminated loop, which also requires a conditional iteration test

According to the above description, static types can be used to trace the entire root set and live heap data with conditional tests and tags in only a few places. This should give an efficient tracing of live memory, in particular for records. Since only tagged sum types and the exception type may be recursive, the tracing functions can be inlined entirely, except at the occurrences of tagged sum types and the exception type.

Due to the amount of inlining described here, stack space is only used in the following cases when tracing the heap: 1) an activation record for each call to each stack frame's tracing function and 2) an activation record for each occurrence of either a tagged sum type or an exception type in the runtime heap value with the type of variable being traced. This could become expensive when tracing long lists. It could also become expensive when tracing deep stacks, such as performing garbage collection in a non-tail-recursive function, at a point deep in its iterations. It might be possible to avoid some of the activation records for tagged sum types with at most one or two children by using e.g. pointer-reversal, but I have not investigated this. It might also be possible to avoid the activation records in the stack tracing implementation sketch in figure 8.7, but not in the one from figure 8.8, which is, however, the one we will need for incremental tracing of major collection cycles. Limiting the stack space for the heap tracing may therefore be an important point to improve on in the future.

Notice that the tracing described here traces shared memory objects multiple times. Whether this is good or bad may depend on the memory-management method chosen and on the program being compiled. For copying collectors or generational collectors using copy collection for young objects (the nursery), this would surely be a problem, since shared objects would then be duplicated. For mark-sweep-style garbage collectors, where tracing only has to mark what is live, the additional traversal is likely not too costly, since the alternative of placing and checking tags on all objects would cause additional conditional branching in the trace, thereby likely reducing the performance. When implementing an Immix-style garbage collector, the overhead in additional tracing for shared memory objects should not be a significant problem. A degenerate example is if a single data structure occupies the entire heap and, say, one hundred local variables, all located at different stack offsets, each contain a pointer to this data structure, then the entire memory would be traced one hundred times, except if parts of it contained references or arrays. Another degenerate example is if a sequence of heap objects are organized such that each object in the sequence has two pointers to the next object, then the heap sharing, and thus tracing, would become exponential. This is evaluated later with a program shown in section 9.2.2 and is found to be an actual issue; more will be said about it for the evaluation. This is not an issue for references or arrays, since due to their tracing marks for avoiding cycles, they are each traversed only once.

One challenge in having the tags at references and arrays is how to efficiently locate and clear those tag bits in the entire heap before a root-set trace. It could be implemented by maintaining two remembered sets with pointers to all references and arrays, respectively, in the entire program. These remembered sets could be used for clearing all mark bits in those memory objects, before the trace is initiated. We could also use a full-heap object-bitmap for tagging, instead of smuggling bits into heap objects directly, where only the tags for references and arrays then happen to be used. Such a bitmap is used in [Blac08]. If this solution is chosen, then it might also be worthwhile to consider placing a tag at tagged sum types in the heap, which could avoid some of the duplicate heap traversal for shared objects. The tagged sum types already require a conditional branch for determining which tag is stored

in memory, except for tagged sum types with only one constructor tag, so the extra conditional branch here is possibly less prohibitive. It would be a good idea to evaluate how performance is affected by having, respectively not having, a mark bit at tagged sum types for the heap tracing, but this will not be done in this project. Going in the opposite direction, the tracing function for tracing a tagged sum type could even be stored as a pointer in the heap-allocated sum type, which would eliminate the conditional check, at the cost of requiring more memory for tagged sum types, which could, however, become quite prohibitive for e.g. lists. References and arrays could also always be allocated into a separate known set of memory areas, which would make it easy to clear their mark bits. This will, however, not give the locality of reference, which the mark-region method otherwise offers. All in all, there are several trade-offs to be made here, most of which would require evaluation, in order to determine what works best in practice.

If we were implementing a two-generation garbage collector, with a copying collector as the nursery and some other collector, e.g. the Immix-style mark-region collector, for the mature space, generating statically typed tracing functions could become more challenging or less attractive. The tracing functions would somehow need to determine whether to do copying (for the nursery) or marking or whatever (for the mature space), which would likely require conditional branching. This problem will, however, not be investigated in any more detail, since the in-place generational method using sticky mark bits from [Demm90] will be used, as also proposed in [Blac08].

The table in figure 8.2 from a few sections ago measures the code size for a single specific program with with and without generated tracing functions. This shows that the size-increase is currently somewhat prohibitive (currently more than a factor of five). As we will see later, the currently code, however, is larger than what is presented here, since it incorporates the code for generational collection and incremental collection. Worse, it even generates the tracing code for each stack frame twice, due to a code-duplication issue, to be seen later.

8.5 Precise Garbage Collection Points

With the method of tracing live memory objects from the previous section, the generated typed tracing functions are installed as function pointers on the call-stack and are able to trace all root pointers from all stack frames *underneath* (in memory-order this is actually above, since the stack grows downwards) the currently executing function. This makes it an obvious idea to only consider calling the memory tracing and the garbage collector at function entry points. This is also what is mentioned to have been done in the article [Hall02] in section 4.4. In [Hall02] section 6 they consider it future work to improve on this and I have not investigated whether they have improved on this since then and if so, how they might have done this.

In order to initiate garbage collection at precisely the points where memory allocation occurs, or rather, is just about to occur, it would be necessary to trace live memory pointers within the currently executing function, in addition to the live memory objects reachable from previous stack frames. This might seem like extra work and with a direct implementation of this idea, it would be, at least implementation-wise. There is, however, an elegant solution. In the compiler, just prior to code generation (also prior to generation of the typed stack tracing functions), a simple program transformation is made in the $FILM_L$ intermediate language. The transformation consists of two simple parts:

1. Declare a non-primitive function (currently named `___CheckGcAndMemAlloc`) in the beginning of the program, which first checks if memory should be garbage collected and if so, does that. It allocates the memory after that, by calling the primitive function currently named `___MemAlloc`

2. Convert all primitive calls to the memory allocation function `___MemAlloc` in the rest of the program into non-primitive calls to the function declared in step 1

Since the function declared in step 1 is not a primitive function, the typed stack frame tracing functions are generated at all call-sites to this function, as for any other non-primitive function. If garbage collection is to be initiated in the declared function `___CheckGcAndMemAlloc`, it will be initiated before any other memory pointers become live, since checking for garbage collection initiation is the first thing it does, which does not in itself require allocating any heap memory.

Implementing the checks for whether garbage collection should be initiated or not in this way avoids having to make the garbage collection initation check at every function entry, such that it is only performed precisely when needed. It also avoids having to determine in which functions memory is allocated and it avoids having to generate extra tracing code before each memory allocation call. Hence, it is both simple to implement and seems to give good and precise garbage collection initiation points.

This method has the overhead that it requires an extra call to the function `___CheckGcAndMemAlloc` when allocating memory. The overhead seems to be small in practice though. An alternative is to change step 2 in the method described above, such that it replaces the primitive calls to `___MemAlloc` with the code from `___CheckGcAndMemAlloc` inlined, except for the actual garbage collection, which could then be generated as a non-primitive call. Then there is only the overhead of a function call whenever garbage collection actually has to be performed. This optimization has however not been implemented. Without this optimization, it is easy to replace the garbage collection initiation policy and to replace the memory allocation function with `malloc` without changing the compiler-generated code, which is useful for the purposes of measuring performance in this project.

8.5.1 Memory Allocation Function Types and Bypassing $FILM_L$ Type-Checking

The previous section describes only one function called `___MemAlloc` and only one declared function called `___CheckGcAndMemAlloc`. In the monomorphic type-system of the $FILM_L$ language, there is actually a `___MemAlloc` function for each type of memory object allocated. The names of these functions are only guaranteed to have the prefix `___MemAlloc`. To make an implementation which would allow correct type-checking, it would actually be necessary to generate a corresponding `___CheckGcAndMemAlloc` function for each `___MemAlloc` function. This is simple to do, but to complete this project on time, this was not done. Instead, the $FILM_L$ type-checking was just not performed after the transformation described in the previous section. The program transformation is sufficiently simple that it worked without problems the first time, so this was not an obstacle to debugging.

8.5.2 Stack Tracing Function Optimizations

If the generated stack tracing functions are generated naively, some duplicate tracing is likely to occur. This section describes two simple optimizations, which have been implemented.

When determining which variables are live and should be traced in the stack frame, this happens at the points just before non-primitive function calls. If we trace all heap pointers of the arguments passed to the function, they will likely be traced again in the stack frame of the called function, which is redundant work. This can be avoided, by not tracing variables which are only live due to being passed as function arguments. There are, however, two things to be very careful about here:

1. If the argument variables are also live in code following the function call, then those relevant variables should still be treated as live and thus be traced. The reason is that, if the arguments are not used by the called function, that function will not treat those parameters as being live and therefore not trace them. This is illustrated by a small example in figure 8.9

2. This optimization requires that when entering a function, all live heap pointers passed in parameters must always be traced before garbage reclamation is done. With the method described previously in section 8.5, this is always the case, since the only place were garbage reclamation is done is in the function `___CheckGcAndMemAlloc`, which does not have any parameters with heap pointers. If garbage collection had been initiated at function entry points, it would have required extra implementation work to generate code for tracing these parameters

When the above optimization of not tracing function arguments needlessly has been implemented, there still may be several live variables, which refer to the same heap pointer stored in the same stack slot. One final optimization is therefore to avoid duplicate tracing of the same stack slot in the generated tracing functions. This can be implemented by sorting the stack slots to be traced according to their offsets and removing duplicates. This optimization is always valid and removes some redundant work in the heap tracing. A desirable order of tracing the variables in the stack slots may also be chosen at this point, e.g. to favour good cache behaviour, but this has not been investigated.

```
(* Assume that q is defined with heap allocation prior to this piece of code *)
letfuns
  fun f(a : [{f1 : int()}]c) : [word()]c =
    (* ___CheckGcAndMemAlloc may trigger garbage collection *)
    let m : [word()]c = ___CheckGcAndMemAlloc(4) in
    M[m + 0] := 5 in
    return m
in
let result : [word()]c = f(q) in
return q
```

Figure 8.9: Example of why variables passed as function call arguments must be kept live, if they are used after the call, when trying to optimize stack-variable tracing of heap pointers. The function *f* is called with the argument *q*, which is the parameter *a* in *f*. The function *f* does however not use *a*, meaning that it is not treated as live. If memory is reclaimed in the function *__CheckGcAndMemAlloc*, then the value held in *q* will be reclaimed at that point, which would turn *q* into a dangling pointer, if it is not traced as live in the tracing function generated before the call to *f*

8.5.3 Controlling Garbage Collection Initation

The previous sections describe that the function `___CheckGcAndMemAlloc` decides whether memory allocation should be performed or not. It does this by calling a primitive function, currently called `___MrShouldReclaimGarbage`, which returns true if garbage collection should be done and false otherwise. This could implement any policy desired, but only a simple policy has been implemented, which, however, should be quite a sensible policy. The chosen policy is described in detail later in section 8.7.1.

8.6 The Immix-Style Mark-Region Garbage Collection Algorithm

The garbage collection algorithm is based on the Immix mark-region method from [Blac08], which was introduced earlier in section 6.9. This section describes some specific details of the implemented algorithm, including information on how it differs from the algorithm in [Blac08].

8.6.1 The Internal Heap Manager

An internal heap manager has been developed, which is quite simple and tailored to the needs of an Immix-style garbage collector. It allocates memory in blocks of 32 kilobytes (32,768 bytes). Each of these blocks are allocated and freed with the POSIX funcions `mmap` and `munmap` on Linux. They would likely have to be allocated with functions like `VirtualAlloc` and `VirtualFree` on Windows (Win32), but that would be a matter of replacing just those two function calls, each of which are called in only one place in the implementation. These two functions constitute the external heap manager.

The garbage collection algorithm, as it is currently implemented, requires each block to be aligned to 32-kilobyte addresses. The POSIX functions `mmap` and `munmap` only align to the operating system's page size, which is typically 4096 bytes, which is not good enough. The way that I have chosen to achieve the 32-kilobyte alignment, is by allocating 16 aligned contiguous blocks at a time, where space for 17 blocks is requested from `mmap` (i.e. slightly more than half a mega byte in total). The start of the first of the 16 blocks is chosen as the next aligned address after the address returned by `mmap`, meaning between one page and one whole block after the address returned by `mmap`. Specifically, if `IHM_BLOCK_SIZE` is defined to be 32768, then the following C-expression returns the aligned address from the address `addr`:

```
((char *)(((int)addr) & (~(IHM_BLOCK_SIZE - 1)))) + IHM_BLOCK_SIZE
```

When the alignment is done in this way, there is always at least one page of unused memory before the first block. The 8 bytes immediately before the first block is used for storing the unaligned start address returned from `mmap` and the requested number of bytes, such that the allocated chunks of memory can be conveniently freed by `munmap` later on, even when only remembering the aligned address of the first block. Freeing up unused chunks of such 16 contiguously allocated blocks is however currently not implemented, so the peak amount of memory used is currently retained until the running program ends.

The 32-kilobyte blocks are maintained in three lists: The list of free blocks, the list of unavailable blocks and the list of recycled blocks. Whenever a chunk of 16 aligned blocks has been allocated, as described above, each of those blocks (regions) are added to the list of free blocks. No multi-thread support has currently been incorporated.

8.6.2 The Memory Region Layout

Each block (region) of 32 kilobytes (32,768 bytes) of memory is organized as follows:

Line-mark bitmap	Block-list pointer (singly linked)	Unused	Usable memory area Contains 255 lines, each 128-bytes
32 bytes (256 bits)	4 bytes	92 bytes	32640 bytes

The first 128 bytes, which would correspond to the first 128-byte line of memory, if the entire block had been used, is used for the line mark bitmap and a pointer for organizing blocks into a list. The internal heap manager thus maintains a few singly linked lists of such blocks. Each such list is formed as illustrated in figure 8.10.

8.6.3 An Overview of the Mark-Region Algorithm

This section briefly describes the mark-region algorithm in terms of the tetracolour abstraction and the three lists: the unavailable list, the recycled list and the free list.

The list of free regions contains off-white regions. During-steady state allocation, the list of recycled regions contains regions whose used lines are black and whose free lines are off-white. During collection,

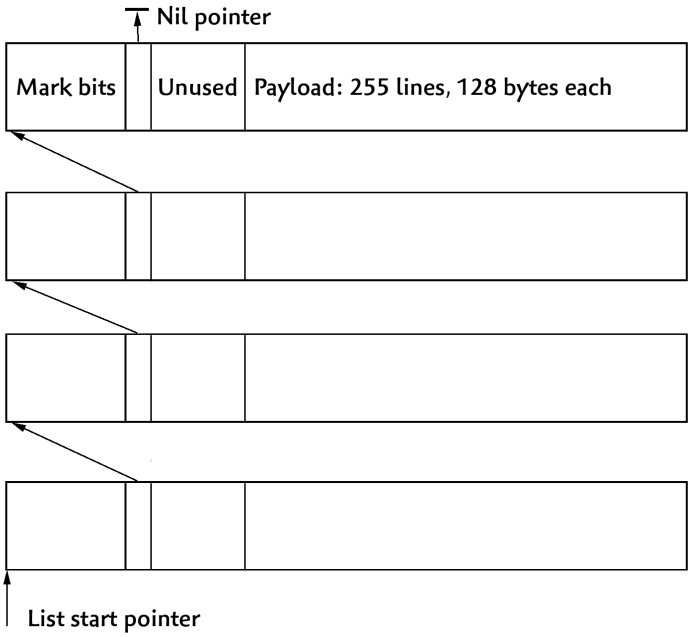

Figure 8.10: An illustration of how allocated individual region blocks are used to form a list, in order to keep track of them

no recycled regions are assumed to exist. Unavailable regions are black during steady-state alloction. During collection, the unavailable regions are initially marked as white, by clearing the mark-bits in the line-mark bitmaps. Tracing then marks lines as grey or black; the algorithm does not care about which of these two colours a line has, so let's call them non-white.

The reclamation traverses the unavailable list and keeps all regions containing only non-white lines in that list, at least in the simple case, which is currently implemented, but this depends on the recycling policy, described in slightly more detail later in section 8.6.5. The reclamation moves all regions containing only white lines to the free-list, where they become off-white. Regions containing a mixture of white and non-white lines are moved to the recycled list, where the non-white lines are then considered black and the white lines then considered off-white.

8.6.4 Marking a Byte of a Memory Object in the Line Mark Bitmap

In the current implementation, the basic computation for marking a bit in the line-mark bitmap for a given byte of a memory object is performed by a sequence of thirteen x86 CPU instructions. In addition to this basic computation, it is necessary to first allocate four registers and perform any spilling required to achieve this.

A function for computing the machine-word address in the line-mark bitmap and a bitmask for marking the appropriate bit is shown as Standard ML code in figure 8.11. An example of a corresponding sequence of x86 CPU instructions generated for performing the same computation is shown in figure 8.12. The Standard ML code does however not perform the bit-wise OR to actually mark the bit in the line-mark bitmap, but the x86 assembly instruction sequence does. Using the API from section 8.3, the bit-marking could be done in Standard ML by the expression:

```
UnsafeMemoryAccess.write32(
```

```
  lineMarkWordAddr, 0,
  Word32.orb(UnsafeMemoryAccess.read32(lineMarkWordAddr, 0), bitMask)
)
```

It would however require a better code-generation algorithm than the one I have currently implemented, mostly in relation to the register allocator, to translate this into the single x86 assembly instruction which is the last instruction in figure 8.12.

```
(* Direct computation of line-mark word and bit *)
(* Requires computation of the constants mrLineMarkBitmapMask,
   mrLineMarkByteMask, mrLineMarkWordShift, mrLineMarkBitShift
   and mrLineMarkBitmask, which is not shown here.
   They depend on the chosen block (region) size and line size *)
fun computeLineMark addr =
      let
        (* The address of the line-mark bitmap *)
        val lineMarkBitmapAddr = Word.andb(addr, mrLineMarkBitmapMask)

        (* The word to mark at within the line-mark bitmap *)
        val word =
              Word.>>(
                Word.andb(addr, mrLineMarkByteMask),
                mrLineMarkWordShift
              )

        (* The actual final word to mark at *)
        val lineMarkWordAddr = lineMarkBitmapAddr + word * 0w4

        (* The bit to mark in the word *)
        val bitNumber =
              Word.andb(
                Word.>>(addr, mrLineMarkBitShift),
                mrLineMarkBitMask
              )
        val bitMask = Word.<<(0w1, bitNumber)
      in
        (lineMarkWordAddr, bitMask)
      end
```

Figure 8.11: The computation in Standard ML of the machine word address in the line-mark bitmap and a bitmask for marking the appropriate bit from a given address of a memory object. The code shown here does not perform the bit-wise OR to actually mark the bit in memory

8.6.5 Testing Line-Mark Bitmaps

When the list of unavailable regions is scanned for reclamation, each region is classified into either free, recyclable or unavailable.

```
movl %edx, %ecx
andl $-32768, %ecx
movl %edx, %eax
andl $32767, %eax
shrl $12, %eax
sall $2, %eax
addl %ecx, %eax
movl %edx, %ecx
shrl $7, %ecx
andl $31, %ecx
movl $3, %esi
sall %cl, %esi
orl %esi, 0(%eax)
```

Figure 8.12: The x86 assembly instruction sequence for marking a bit in the line mark bitmap from a given address of a memory object, assumed to initially be in the register *edx* (and retained there after the computation). The instruction sequence shown does not include register allocation, which in the current implementation is done before this sequence, since the entire sequence is currently hard coded and generated inline by the compiler. The instruction sequence is in GNU Assembler (GAS) format, where the source operand is written first and the destination last. The immediate constants -32768, 32767, 12, 2, 7, 31 and 3 are always the same as shown here, which is for a block size of 32768 bytes and a line size of 128 bytes. The constant 3 gives the conservative line marking, as explained in section 8.6.6. Without conservative line marking, this constant would have been 1; the rest of the sequence would be otherwise unchanged

For a region to be free, all bits in the line mark bitmap must be zero. A quick way to test this, which is used in the current implementation, is to notice that it consists of only eight 32-bit machine words. These can be read one at a time and bit-wise ORed into a single 32-bit word. If this resulting word is 0, the region is considered as free. This computation only takes eight memory reads, seven bit-wise OR operations, one equality comparison and a conditional jump. This is therefore a very quick test with only one conditional jump, which is thus able to quickly classify all free regions in the list. If the eight 32-bit words of the line-mark bitmap are read into the variables w0, w1, ..., w7, all assumed to have type `unsigned int`, then the following C-expression computes the condition for the classification:

```
(w0 | w1 | w2 | w3 | w4 | w5 | w6 | w7) != 0
```

The regions which are not classified as free must be classified as either recyclable or unavailable. According to section 3.1.1 in [Blac08], they found that the simple recycling policy of recycling all blocks with at least one free line works best. This is also a policy, for which it is as efficient to classify regions as recyclable as it is to classify them as free, since the test is done very similarly. I have implemented it by bit-wise ANDing the eight 32-bit machine words of the line map and testing that the resulting word has all bits set, i.e. is equal to $32^2 - 1$. There is one caveat though. Firstly, the first line in the region is never used, since the region header information resides in that line. Also, due to the way that conservative marking has been implemented, the first bit in each of the eight 32-bit machine words may be occupied, despite being marked as free, as will further explained in section 8.6.6. Hence, the first bit should not be tested when the eight 32-bit words have been bitwise ANDed together, which is accomplished by performing a bitwise OR with the first bit set, before the equality test with the constant $32^2 - 1$. If the eight 32-bit words of the line-mark bitmap are read into the variables w0, w1, ..., w7 as before, then the following C-expression computes the condition for the classification:

```
((w0 & w1 & w2 & w3 & w4 & w5 & w6 & w7) | 0x1) == (int)(-1)
```

This test then gives the final classification of the block as either recyclable or unavailable.

8.6.6 Allocating with Line Mark Bitmaps

It is stated in section 3.3 in [Blac08] that they use conservative line marking, which implicitly always marks the first line in a hole in the line-mark bitmap. It is stated that this is done during allocation into free lines, by skipping the first free line in each hole. I am not sure if they actually implement it in this way, but it is not that way that I have implemented it. A reason why they might have implemented it in that way might be that they state that they only mark objects during tracing and create the line-mark bitmap later. My proposal here here would not be compatible with that.

The way I have implemented line-marking is by always marking both the line bit to actually be marked, as well as the line bit immediately following it. This is a matter of bit-wise ORing with the shifted constant 3, instead of the shifted constant 1. There is one caveat here though: the lines are marked by shifting a bit to the appropriate position in each of the eight 32-bit words that the line-mark bitmap consists of. This means that when the constant 3 is shifted to mark the bit representing the last line in such a word and the following bit, the extra bit is actually shifted out of the word and nothing thus marked. Therefore, the first bits in the last seven out of the eight 32-bit words in the line-mark bitmap are not reliable for achieving conservative line marking in this way. This is, however, not a problem in practice; the classification between unavailable regions and recycled regions and the end of section 8.6.5 took properly care of this.

It is stated in [Blac08] that conservative line marking significantly speeds up marking of small objects, but it is not described how. The reason for this is that without conservative line marking, it would be necessary to mark both the start and the end byte of all objects. For objects smaller than one line this would mean that lines corresponding to two different bytes would have to be found and marked. On the other hand, when an extra line is always implicitly marked in the line-mark bitmap, only the first byte of every line of each object have to be marked. Thus, only half as many mark operations have to be performed for small objects (i.e. objects of at most the size of one line). Since setting a mark takes at least 13 instructions in the current implementation, as shown in figure 8.12, this is quite a savings.

8.6.7 In-Place Generational Collection

The method as described so far is a full-heap collector without any support for incremental or generational collection. It is fortunately very easy to extend the method to become an in-place generational collector, using the sticky-mark-bits method from [Demm90], as also used in [Blac08].

By implementing the sticky-mark-bits algorithm, it becomes possible to do partial heap collections of only the most recently allocated objects, in addition to the ordinary full-heap collections. Implementing the sticky-mark-bits algorithm consists of two simple modifications:

1. The mark bits are not cleared in all unavailable regions prior to the collection, as is normally done for full heap collections

2. Marking is only performed for the memory objects allocated since the last collection, which could have been either a partial or a full collection. This is achieved by modifying the stack tracing functions to support tracing only the stack frames which were not traced in the previous collection

The first of these modifications is obviously trivial to implement, so only the second modification will be described in more detail here.

The key part of the tracing-function algorithm to modify is to check whether to call the next stack frame tracing function or not, depending on whether it has already been called since the last collection.

For the updated tracing functions, we need to be able to perform both a full heap collection and a partial heap collection of only the most recently allocated heap objects. This will be distinguished by passing a bit-mask parameter through all the tracing functions. This is the parameter named `clearMask` in figure 8.13, which shows a sketch of the updated tracing function, compared to that which was shown earlier in figure 8.8. If the first stack tracing function is called with a `clearMask` argument of `0wxfffffffe` (assuming that pointers are 32-bits wide), a full stack tracing is performed. If `clearMask` is `0wxffffffff`, only the stack frames created since the previous collection are traced. A more portable way of specifying these values is to specify `0wxfffffffe` as the bit-wise negation of `0wx1` and `0wxffffffff` as the bit-wise negation of `0wx0`.

As can be seen, the modifications, as compared to figure 8.8, for implementing the sticky mark-bits tracing for full and partial in-place generational heap collections, are modest.

It is possible to implement this in-place generational heap-tracing by tracing the stack frame before making the call to the next stack-frame-tracing function, as was done in figure 8.7, but this will not be shown and has not been implemented.

Notice that in the code in figure 8.13 the tracing code is generated twice, which is code-duplication due to the way that $FILM_L$-expressions are defined; no code can follow an if-expression without getting inlined into both branches. This could be solved by declaring a local function which is called twice, but this has not been implemented. Instead, I realized that the tracing in the last tracing in the else-branch may not even be necessary, since in this case, the stack-frame is already marked as traced, meaning

```
fun stickymarkbits_frametracer(stackPtr : [word()]c, clearMask : word()) : int() =
    let nextFramePtr : [word()]m = ___UnsafeMemoryAccess.addradd(stackPtr, <stack frame size>) in
    let readTracerFuncPtr : word() = M[nextFramePtr + 0] in
    let nextTracerFuncPtr : word() = ___Word.andb(readTracerFuncPtr, clearMask) in
    let maskBit           : word() = ___Word.andb(nextTracerFuncPtr, 0w1) in
    if maskBit ___Word.= 0w0 then
      let markedTracerFuncPtr : word() = ___Word.orb(nextTracerFuncPtr, 0w1) in
      M[nextFramePtr + 0] := markedTracerFuncPtr;
      let u : int() = ___CallFunctionPtr(nextTracerFuncPtr, nextFramePtr, clearMask) in
      <Trace the variables of the stack frame with base at stackPtr>
      return 0
    else
      M[nextFramePtr + 0] := nextTracerFuncPtr;
      <Trace the variables of the stack frame with base at stackPtr>
      return 0
```

Figure 8.13: Sketch of a stack frame's tracing function as $FILM_L$ code, when supporting the tracing required for sticky-mark-bits in-place generational collection, where the trace function can trace for either a full heap collection or for a partial collection of only the most recently allocated heap objects. If the first stack tracing function is called with a *clearMask* argument of *0wxfffffffe* (assuming that pointers are 32-bits wide), a full stack tracing is performed. If *clearMask* is *0wxffffffff*, only the stack frames created since the previous collection are traced. For portability, the *clearMask* argument should be specified as the bit-wise negation of *0wx1*, respectively *0wx0*. As can be seen, the modifications, as compared to figure 8.8, for implementing this are modest

that it would probably by redundant to trace it again. Something similar is the case for the incremental implementation in the next section, but when I tried removing the other trace, the tracing calls started crashing, so something needs further investigation here.

When using this generational tracing, write-barriers are required, in order to make sure that already-traced references or arrays (which are black) updated to point to new memory objects (which are white) have their updates tracked and properly handled. There was no time to extend the compiler to support references or arrays, so this will not be described. Write-barrier methods were thoroughly described in section 4.11.4.

The final issue to consider is when and how often to perform partial versus full heap collections. This policy needs further work and the current suggestions are explained in section 8.7.2.

8.6.8 Incremental Full-Heap Collection

The in-place generational collection from section 8.6.7 makes the full-heap collections rarer, which helps in making long pause times rarer to improve the performance. It does, however, not limit the length of the long pause times. On the other hand, it should not increase the longest pause times, since pause times during collection is determined by the amount of live data, not by how frequently garbage collection is triggered or whether some collection cycles are only partial.

Extensions to the sketch function from figure 8.13 are shown in figure 8.14. These are the required changes to implement incremental full-heap collections on top of the sticky-mark-bits algorithm. There was however not enough time to get this properly debugged and up and running, so it may have mistakes, although debug-prints verified that its tracing behaviour seems sensible. This function is used in all the experiments and it works correctly for in-place generational collection and full-heap collection. The

extension only involves adding extra checks, such that root-set tracing (on the backwards retrace of the function calls) only continues as long as a primitive function, which is named ___MrShouldStillTrace, returns true.

The return value of the function ___MrShouldStillTrace could be based on either a maximum time-budget or on how many times it has been called since the tracing was initiated, corresponding to how many stack frames it has already traced. Suggested policies for this are described in section 8.7.3. This policy is important, since e.g. if much memory is allocated rapidly in a non-tail-recursive function, the collector may not be able to "keep up" with the mutator on collecting garbage, if the incremental collections are interrupted too quickly or not triggered frequently enough.

```
fun incremental_stickymarkbits_frametracer(stackPtr : [word()]c, clearMask : word()) : int() =
    let nextFramePtr : [word()]m = ___UnsafeMemoryAccess.addradd(stackPtr, <stack frame size>) in
    let readTracerFuncPtr : word() = M[nextFramePtr + 0] in
    let nextTracerFuncPtr : word() = ___Word.andb(readTracerFuncPtr, clearMask) in
    let maskBit           : word() = ___Word.andb(nextTracerFuncPtr, 0w1) in
    if maskBit ___Word.= 0w0 then
      let u : int() = ___CallFunctionPtr(nextTracerFuncPtr, nextFramePtr, clearMask) in
      let stillTracing1 : bool() = ___MrShouldStillTrace() in
      if stillTracing1 ___= true then
        let markedTracerFuncPtr : word() = ___Word.orb(nextTracerFuncPtr, 0w1) in
        M[nextFramePtr + 0] := markedTracerFuncPtr;
        <Trace the variables of the stack frame with base at stackPtr>
        return 0
      else
        M[nextFramePtr + 0] := nextTracerFuncPtr;
        return 0
    else
      let stillTracing2 : bool() = ___MrShouldStillTrace() in
      if stillTracing2 ___= true then
        M[nextFramePtr + 0] := nextTracerFuncPtr;
        <Trace the variables of the stack frame with base at stackPtr>
        return 0
      else
        M[nextFramePtr + 0] := nextTracerFuncPtr;
        return 0
```

Figure 8.14: Sketch of a stack frame's tracing function as $FILM_L$ code, when supporting the tracing required for incremental full-heap tracing as well as sticky-mark-bits in-place generational collection. Root-set tracing only continues as long as the function *___MrShouldStillTrace* returns true. This could be based on either a maximum time-budget or on how many times it has been called since the tracing was initiated, corresponding to how many stack frames it has already traced. As can be seen, the modifications, as compared to figure 8.13, for implementing this are modest. This is the function used in all the experiments. It works for full-heap collection as well as in-place generational collection and it seems to have the correct tracing behaviour (verified by debug-printing) for incremental collection

The issues with code-duplication mentioned in the previous section (section 8.6.7) still apply here. Write-barriers for references and arrays are also still required, as mentioned in the previous section.

8.7 Garbage Collection Policies

This section describes the garbage collection policies, which have been implemented in the current version of the memory manager.

8.7.1 Garbage Collection Trigger Policy

The section 8.5.3 from earlier mentioned that a simple policy had been implemented for deciding when to trigger garbage collection, implemented by the return value of a function named `___MrShouldReclaimGarbage`. This section describes the policy in the simple setting of performing non-incremental and non-generational (i.e. full-heap) collections. The policy described is work-based.

The article [Hall02] mentions in section 4 that they garbage collect a region heap when the size of its region free-list becomes less than $1/3$ of its total size. They also state that after collection, they make sure that the number of region pages is at least three times the size of to-space, which they refer to as having a heap-to-live ratio of 3. This corresponds well to a heap occupancy (or residency) of 3, which is defined as the ratio between the live (resident) amount of memory to the amount of memory allocated by the heap manager. The Immix article [Blac08] makes a thorough analysis of the impact of varying the memory residency from factors of 1 through 6. I have not tried to implement an exact measure of liveness, but instead only a (possibly very) conservative approximation L to the amount of live memory. Also, I relate the conservatively live amount of memory L to the amount of memory A requested by the mutator through allocation, not to the amount of memory allocated by the heap manager.

The conservative estimate of live memory L is computed as the number of memory regions which are in use (either unavailable or recycled) multiplied by the number of bytes that a region can at most contain. However, L is limited to a minimum value, which is the number of bytes which can be held in one region, to ensure that it is never zero and has a sensible minimum value. The amount A of memory requested by the mutator through allocation is computed by adding the number of bytes allocated by the mutator since the last collection to the conservative number L from the previous collection. The number of bytes allocated is tracked precisely for statistical purposes, so this number is precise.

Garbage collection is performed if A is more than three times higher than L. This is described by the relation $3L < A$. The return value of the function `___MrShouldReclaimGarbage` is computed such that it returns true if the relation $3L < A$ is true and all recycled memory blocks have been spent. Garbage is thus never reclaimed as long as recycled blocks remain. Remember that L was taken to be at least the number of bytes in a single region. When computing A, it also uses this compensated value of L. Finally notice that, the value L is from the previous collection and A is partially computed from that L.

It would be possible to compute the precise number of live bytes, e.g. by summing up bytes in the tracing functions, where many byte sizes are even statically known and therefore can be generated as constants in the code. However, this is currently not done. The summing of bytes would also take slightly more computational work during the tracing and generate slightly more tracing code, although it should not be prohibitive. More importantly, the current tracing, which traces shared heap values mutiple times, will not be good enough, so it would be necessary to tag all structured data values, i.e. also records and tagged sum types, which is one of the things that have specifically been avoided.

I have considered resetting A when the last recycled block is spent during allocation, since all the recycled blocks were conservatively estimated to be full. This could probably improve the precision of the estimate considerably, but there was no time to investigate this. The current estimates do seem to have problems though, so further investigation is needed on this.

It would also be possible to use a time-based scheduling policy, such as triggering garbage collection

if has been is at least e.g. 500ms since the last collection. This is argued in [Baco03] to be a superior collection triggering policy for real-time purposes. When extending to generational partial heap traces and, even more importantly, incremental collections, time-based scheduling becomes more important. These policies will be discussed in the following sections. The problem with work-based scheduling is that even though they may keep pause-times short, they do not necessarily give good mutator utilization, meaning that collections may end up being triggered often during some periods and rarely in other periods Time-based scheduling has however not been investigated much and is considered important furture work.

8.7.2 Sticky Mark Bits Partial Collection Frequency Policy

When enabling in-place generational collection using the sticky-mark-bits algorithm, a policy needs to be defined to choose between performing a full heap collection or a partial heap collection. This is still different from performing collections incrementally.

The policy mentioned in the article [Demm90] for sticky-mark-bits generational collection is to perform a collection every time around 100 kilobytes have been allocated. I have tried this and it does *not* work well in practice, at least not with the large amounts of memory allocated with the test-programs that I have tried. One of the programs (I cannot recall which) performed more than 2 million partial collections with this policy and thereby ran in more than ten seconds, instead of less than two seconds as with only full-heap collection.

The problem seems to be using a constant amount of kilobytes, rather than making the policy work-based. If collecting when some specific fraction of the amount of live bytes from the previous collection has been reached, things seem to work better. This is the policy currently used. I currently trigger a partial collection, if the amount of memory allocated since the last *partial or full* heap collection is more than $1/3$ of the amount of live memory since the last *full* heap collection. Performance varies when changing the fraction $1/3$ and this setting is not very well tuned. Nor are the liveness estimates very accurate, so requires further work.

One could also consider triggering a full collection e.g. only every 16th time a full collection would otherwise be triggered and instead trigger partial collections the other fifteen times. I have not tried this and it does not seem to be a good idea either, unless e.g. the full heap collections are triggered by a time-based policy. One could also consider a time-based triggering of partial collections, while the work-based trigger only applies to full heap collections. As stated, finding a proper policy here requires more work.

8.7.3 Incremental Collection Policy

When considering policies for performing incremental collection, the policy has to interoperate with both of the policies for when triggering garbage collection and when performing full versus partial collections of the heap. The incremental collection policy in this section is mostly concerned with how to terminate a collection in progress, in order to make the whole collection cycle incremental, hopefully without too long pauses. There are actually four scenarios: 1) a full-heap collection, which performs the whole collection cycle by one collection, 2) a partial-heap collection, which performs a whole collection cycle for a part of the heap in one collection, 3) a full-heap collection, which is only part of a collection cycle and 4) a partial-heap collection, which is only part of a collection cycle. In the implementation, this is currently not properly implemented and I did not have much time to investigate this policy.

I can think of a few simple ways of terminating an incremental collection. One is to terminate it after having traced some number of stack frames or some number of bytes. Terminating based on a

number of stack frames does not seem like a good idea, since one stack frame could potentially take a long time to trace, as will actually be the case in one of the test programs for the evaluation. Using a time-based policy, such as terminating the collection after a time period corresponding to the maximum desired pause time as elapsed, seems more suitable here. The implementation of incremental collection was not completed on time, so I did not have time to investigate this policy in practice at all.

One important problem to mention in terminating incremental collections is that the collector may not be able to "keep up" with the mutator, if much memory is allocated rapidly in a non-tail-recursive function. Allowing tracing for e.g. 20ms, which corresponds to skipping one frame at a 50-frames-per-second frame-rate in a game, should however allow for a significant amount of tracing on a modern computer. All in all, there may be several issues to consider in determining this policy.

8.8 Extensions to the Implementation

This section describes some very relevant extensions, which could be made to the currently implemented memory-management system.

8.8.1 Handling Large Objects

There is currently no special handling of large objects in the implemented memory-management system. This specifically means that it will never be able to allocate memory blocks larger than 32 kilobytes (minus 128 bytes). This limitation is important to address.

The article [Blac08] suggests handling objects larger than 8 kilobytes in a separate large object space (LOS). This could be implemented by allocating such large objects separately with some extra space for header-information, to allow the large memory areas to have a mark-bit and be kept in a separate list. This list could have its unused objects reclaimed when the coarse-grained reclamation of regions is performed. Clearing these mark-bits would have to be done whenever line-mark bitmaps are cleared.

The only objects which could have such large sizes are records and arrays. Records would be statically determined to have that size, so runtime checking of whether to allocate, mark and traverse a specialized large object memory layout can be restricted to arrays, due to the statically generated heap-tracing functions.

Hence, there does not seem to be any problems in implementing this, but the time did not allow for implementing it and none of the currently working benchmark-programs allocates such objects.

There is, however, a potential problem in relation to inteactive applications, which is that of pause-times. Marking a large (boxed) array of more than 8 kilobytes may give significant pause-times. For records, the number and offsets of their children whould be statically known and they are probably not likely to have a significantly high number of children in practice. Hence, for records, it is either not a problem in practice or it could be detected at compile-time, with appropriate code statically generated for making the tracing incremental. It should be possible to handle large boxed arrays by incremental tracing. It might require a bit of extra information in the memory-object header for large arrays, in order to be able to track progress of incremental tracing, but it should not be too hard to implement.

The implemented mark-region method as such does therefore not seem to have any problems in handling large memory objects, not even for interactive applications, although the current implementation does not support it.

8.8.2 Defragmentation

If the mutator runs for a long time without defragmentation ever being performed, the heap may become fragmented, which may in turn degrade the collector's performance over time, especially for allocation into recycled regions. This is therefore important to address for a production system.

Defragmentation in [Blac08] is described to be done opportunistically. At the start of each collection, the collector decides wether to defragment or not, e.g. based on fragmentation statistics from the previous collection. The defragmentation is done by selecting regions for defragmentation before starting tracing. Objects in the regions chosen for defragmentation are then evacuated away from those regions and allocated into new regions by the normal mark-region allocation. This is performed in the same pass as the tracing and the evacuation is similar to that of Cheney's copy collector, as implemented by the `evacuate` function in figure 1.1 in section 1.2.2.

It is stated in [Blac08] that defragmentation is only triggered occasionally. For getting an optimized garbage collection in the common case and only paying the price of defragmentation when needed, it may be necessary to store two pointers to tracing functions on the stack frame for each activation record, instead of just the one which is currently stored. This gives some stack-overhead, as well as code-size overhead in the number of tracing functions that have to be generated. Since defragmentation is supposedly done rarely, there may not be a need to optimize on this and simpler ways of implementing it might be possible. The important point is that it is possible to extend the implementation to support fragmentation without problems; how it is best done may require investigation.

8.8.3 Object Bitmaps Versus Line Bitmaps

The current implementation of the garbage collector uses only line-mark bitmaps. In [Blac08] they also use object marking, which could be done either by object-mark bitmaps or bit-marks in the objects, as suggested in this thesis for references and arrays. The problem with object marking in the current scheme is that the entire heap is not traversed in an initial sweep phase, which is one of the advantages performance-wise, but makes it hard to clear the object-mark bits. Using an object-mark bitmap should solve this problem. Investigating the performance of this versus the current line-mark bitmap is suggested for future work.

8.8.4 Marking Tagged Sum Types and Possibly Records

When considering to use object-mark bitmaps, adding tags to tagged sum-types and records becomes possible, without having to worry about clearing those tags. Performance-wise, it takes a little time to test a tag during the heap-traversal, since it introduces a potentially expensive conditional test. On the other hand, it can eliminate the problem with long tracing times of data structures with exponential heap-sharing, as mentioned earler, which is a problem in the current implementation. This can be considered independently for records and tagged sum-types. I believe that it might be a good idea to have such tags for tagged sum-times, which already have the price of a conditional test of the tag of the type at runtime. For records, however, it might be worthwhile to avoid the tags. Exponential heap-sharing can only be constructed with records to the extent that it is constructed statically in the program, since Standard ML does not have recursive types. It might be possible to statically analyse in which cases tagging for records would be desired or not, but I have not investigated this.

8.8.5 Multi-Threading Support

As described in [Blac08], the mark-region method works well with multi-threading support. It does not even require much synchronization between threads. For instance, recycled blocks can be retained locally to a thread, such that only when entirely free blocks are reclaimed or requested, is it necessary to synchronize with the global list of free blocks. Each thread only *allocates* into its own local memory area, although allocated memory may be shared between threads. Only the collection is done for all threads at once. In the Immix system from [Blac08], this can run in parallel in multiple threads on different processors. Races are allowed when marking objects, since the worst-case consequence of this is that some objects may be marked and traversed more than once.

As argued in [Auer07], it took a large engineering effort to port their Metronome implementation from [Blac03b] to a multi-threaded symmetric multi processor (SMP) system. I also expect that this might be the case for this implementation, although the simplicity of the method itself may make this considerably easier than for the Metronome system considered in [Auer07]. How to implement multi-thread support will therefore not be considered any further.

8.8.6 Dynamically Loaded Link Libraries (DLLs)

The implemented method does not seem to require a very specialized binary interface between the mutator and the collector. This should make it fairly easy to extend the implementation to support dynamically loaded link libraries (DLLs), where support in this case refers to both calling but also, in particular, generating such libraries. As with most DLLs, both the compiled and the loaded DLLs would have to be compiled with and for the same system or at least the same binary interface.

One part of the interface is allocation of memory and performing garbage collection. Both allocation and garbage collection can, as is currently done, be factored into a few function calls. Two of these functions are the functions named `___MrShouldReclaimGarbage` and `___MemAlloc` from section 8.5. Some of the inlining optimizations related to those function suggested in that section may, however, not be valid for dynamically loaded link libraries. The current implemention works very will without such optimizations, so it does not seem as if they are crucial for performance. Further, those optimizations may even still be valid in the main-program, which loads and links with other dynamically loaded link libraries (DLLs). The other functions needed for allocation and garbage collection are a function for clearing line-mark bits in the line-mark bitmap, as well as a few policy functions. Calling any of these required functions could be done by dynamically loaded link libraries (DLLs), by exposing those functions from the main program and letting the library perform call-backs to those functions.

The binary interface also consists of the root-set tracing, which is currently done with the generated specialized tracing functions. The root-set tracing is currently triggered by code generated by the compiler, whenever garbage collection is to be performed, which is thus also part of the interface. The binary interface of the root-set tracing functions themselves is, however, very simple and minimal: it consists of storing a function pointer in the stack slot just beneath (in the stack, which is above in terms of memory addresses) the return address pushed to the stack by the x86 assembly instruction `call`. This is a modest requirement on the calling convention, which corresponds to always having the tracing-function pointer as an extra stack parameter for all functions. There is also the restriction that the function pointers must contain addresses which are properly aligned, such that the low bit is available for marking, but this is still a modest requirement. For primitive functions, which do not allocate memory or which prefer to do their own local memory management, there is even the option of classifying calls as primitive calls, in which case they currently follow the ordinary *cdecl* convention, as used for the GNU C Compiler (gcc) on Linux. The *cdecl* convention may vary a bit, depending on

compiler and architecture, but for use with the same compiler and the same architecture, this does not pose any problems. The classification between primitive calls and non-primitive calls, where the latter interoperates with the garbage collection, has to be done statically, which is the only requirement for supporting both kinds of calls in the same program.

In summary, it seems very feasible to extend the implementation to support dynamically loaded link libraries (DLLs). For multi-threaded DLL-support, a slightly more advanced interface may be needed, but it should still be possible to support.

Chapter 9

Evaluation and Experiments

This section describes the experimental evaluation performed on the implemented memory-management system and the results of of that evaluation.

9.1 Experimental Setup

The article [Blac08] has a very throrough and impressive experimental setup. Since the implemented method is similar, it makes sense to use this as a role model for the measurements, although the experiments will not be made as thoroughly as they do in that article. The effect of the individual parts of the implemetation on performance will in general not be evaluated, nor will any comparison with other systems. Hence, only certain performance aspects of the entire implementation will be measured.

When evaluating pause times, only the pause times for garbage *collections* are measured when performing all other measurements. The reason is that, when also measuring pause times of memory *allocations*, which is the other major source of pauses, in addition to collections, the program took several times longer to execute, likely due to the extremely frequent allocations. The pause times when doing collections includes the time to clear the line mark bits, mark the live data and perform the coarse grained sweep of blocks, classifying them as either free, recycled or unavailable. The pause times when doing allocation includes finding a free block of memory, which in turn may require searching for free lines within a region, in case the searched block is recycled.

When measuring the performance, the experiments ought to be done as in e.g. [Blac08], by filtering out the highest value and the lowest value and taking the average of the rest. However, due to time constraints on the project, the measurements are currently just single measurements. All programs are executed in sequence in a makefile and the compiler compiles the program immediately before executing them. Since earlier test runs of the programs had been done just prior to the final executions, the measurements do hopefully not depend too where I generally heavily on caching and program loading. Single executions, however, are generally not very reliable, since they may have high statistical variance, but from experience of running the programs many times, the timings are very indicative of typical executions.

The measurements were performed on a 2.4GHz Intel Duo Core 2 with 3GB memory. Measurements were made for all programs with four different configurations: 1) using only the C-function `malloc` (without `free`) for allocating heap-memory, 2) using the implemented memory-manager's allocation function without ever triggering garbage collection, 3) using the implemented memory manager configured to only perform full-heap collections and 4) using the memory manager configured to use its in-place generational garbage collection. Only one configuration-file for the memory-management implementation

in C was modified for this; the compiler was unmodified.

9.1.1 Suggestions for Other Experiments

The currently implementated system is only evaluated on its own with a few configuration changes. It would be very relevant to compare it with other memory-management methods and implementations. The current compiler-implementation is, however, quite specialized to the implemented mark-region method, especially with respect to the generated typed stack-tracing functions. It would therefore take some effort to implement another method to compare it to, which there was not time enough to do.

One could consider comparing the full system to another compiler, in a similar way as the article [Hall02] compares the ML Kit to SML/NJ in the article . However, this does not necessarily say much about the memory-management system, since the compiler could be very inefficient in one system but have a very efficient memory-management method, while the other system could be opposite of this. Since the current compiler does not perform much optimization and even introduces some overhead in the compilation process, since the implementation has not yet been tuned in any way, it cannot yet be expected to be competitive to other compilers. E.g. my currenly installed version of MLton [MLto07] seems to run the `fib35.sml` benchmark more than 30 times faster than the implemented compiler, where this benchmark is a prime example of a program, for which no dynamic memory management should even be necessary with a decent compilation. Also, the timings and memory usage from [Hall02] show that their compiler uses both less time, despite running on a slower machine, and uses significantly less memory. Hence, no comparisons with other compilers are made.

9.2 Benchmark Programs

The most relevant benchmark programs that I have been able to find, are the ones used in the article [Hall02]. This is a subset of the benchmarks from MLton's website [MLton]. The program `fib35.sml` is used in [Hall02] and here assumed to be like `fib.sml` from [MLton] with the number 44 replaced by 35 and a corresponding modification for the tested result. The program is included in appendix 12.1. The program `msort.sml` is also used in [Hall02] and stated to be a program which sorts 100,000 integers. I found a program named `msort.sml` (and its companion file `msortrun.sml`) in the source code distribution for the ML Kit [MLKit] version 4.3.0. This program only sorts 50,000 integers, so I modified it to sort 100,000 integers and implemented its missing function `upto`, using `List.tabulate`. With these changes, I assume it to be the same as `msort.sml` from [Hall02].

When compiling these programs, they are each prefixed with approximately 1,000 lines of basis library code. All unused parts of that basis library code gets successfully eliminated by the monomorphization phase, which is the first major compilation phase.

Out of these suggested benchmark programs, the programs `fib.sml`, `fib35.sml`, `tak.sml` and `msort.sml` are currently working. All the other programs that I considered (those of the programs from [Hall02] with at most 500 lines of code and which do not have names starting with `kit`) use references, which is not yet implemented in the compiler. I tried compiling the program `life.cxl` from the MLton website [MLton], which I assume to be similar to `kitlife.sml` in [Hall02]. When compiling this, I encountered a bug in the compiler front- and middle-end, which I did not have time to debug. `life.sml` could therefore not be executed either, despite being a program not using references.

All previously available benchmarks will be shown above the line in the performance figures 9.4, 9.5 and 9.6. The programs below the lines were developed specifically for this thesis and are described in the next section.

9.2.1 A Program for Evaluating Stack Usage During Tracing

Since the stack-space usage may be prohibitive for tracing the stack frames in the current compiler-implementation, a small test program for evaluating this has been written. The program is a simple counting loop, shown in figure 9.1.

```
fun count n =
      if n < 100000 then
        (count (n + 1)) + 1
      else
        (print "Successfully completed counting loop\n"; 0)

val result = count 0
val () = if result = 100000 then
           print "The final count was 100,000 as expected\n"
         else
           print "Something went wrong in the counting\n"
```

Figure 9.1: The program *Count.sml*, for evaluating stack-space usage during tracing

This program is not particularly interesting otherwise, since it only computes integers. Hence, for a decent compiler, including future versions of the implemented compiler, there should be no heap allocated values at all.

9.2.2 A Program for Evaluating Exponential Heap Sharing

The current stack-tracing implementation does not try to avoid revisting memory objects which have already been marked. If memory objects are heavily shared, this will result in duplicate tracing of shared objects. A small test program for evaluating this has been written. The program constructs a data structure, which has exponential sharing of memory objects and is shown in figure 9.2. The program contains a counting loop after the data structure with exponential sharing has been constructed, to ensure that the program runs for at least long enough and allocates enough memory that garbage collection will actually be triggered. The last part of the program inspects the constructed data structure and prints a message based on what it sees, to ensure that dead-code elimination would not just entirely optimize way the data structure. I doubt that the latter would be a problem with the current compiler-implementation, but the counting loop is necessary for garbage collection to be triggered.

Three different versions of this program will be evaluated, for constructing exponential sharing of depths 18, 20 and 22, respectively. This is to give a loose indication of the time-complexity of garbage collection for the program. This program might be generally useful for other compilers as well for evaluating memory-manager performance. For other compilers or future versions of the current compiler, it might, however, be worthwhile to replace the counting loop with something which for sure will take time and allocate memory, since the counting loop would likely execute fast and not heap-allocate any memory; it would likely only stack-allocate memory.

9.2.3 A Slightly More Real-Life Example

The program in figure 9.3 is a simple program, which tries to do something more "real-life-like" than the benchmark, though still being compilable with the current system. The program creates 1,000,000 2D triangles with integer coordinates and then transforms them by translating (moving, in geometric terms)

```
datatype exponentialsharing =
  ExlNode of exponentialsharing * exponentialsharing
| ExlLeaf

fun construct (n, node) =
      if n > 0 then
        construct (n - 1, ExlNode (node, node))
      else
        node

(* Create the exponential sharing data *)
val exlSharing = construct(20, ExlLeaf)

(* Do something which allocates a lot of memory, to trigger garbage collection *)
fun count n =
      if n < 100000 then
        (count (n + 1)) + 1
      else
        (print "Successfully completed counting loop\n"; 0)

val result = count 0
val () = if result = 100000 then
           print "The final count was 100,000 as expected\n"
         else
           print "Something went wrong in the counting\n"

(* Make sure that the exlSharing value is not dead *)
val () = case exlSharing of
           ExlNode n =>
             print "Nodes were created\n"
         | ExlLeaf =>
             print "No nodes were created\n"
```

Figure 9.2: The program *ExlSharing20.sml*, for evaluating exponential sharing during tracing

them in the y-coordinate. The memory behaviour is probably not much different from `Count.sml` or `msort.sml`, but at least the amount of live data is larger and the records in the list are heap-allocated, rather than being unboxed integers.

```
fun makeITrigon2d (t : int) =
      ({x = t, y = 0},
       {x = t + 5, y = 3},
       {x = t, y = 5}
      )

fun createITrigons2d n =
      if n < 1000000 then
        let
          val trigon = makeITrigon2d n
        in
          trigon::(createITrigons2d (n + 1))
        end
      else
        (print "Built the list of 2D integer trigons\n"; nil)

(* Create 100,000 2d trigons with integer coordinates *)
val trigons = createITrigons2d 0
val () = print "Returned from creating 1,000,000 trigons\n"

fun translateIPointY distance {x, y} =
      {x = x, y = y + distance}

fun translateITrigonY distance (v0, v1, v2) =
      (translateIPointY distance v0,
       translateIPointY distance v1,
       translateIPointY distance v2
      )

(* Move the 100,000 trigons 25 in y-direction *)
val movedTrigons = List.map (translateITrigonY 25) trigons
val () = print "Moved 1,000,000 integer trigons\n"
```

Figure 9.3: The program *Trigons.sml*, for creating and translating 1,000,000 2D triangles

9.2.4 Suggestions for Other Test Programs

It would be relevant to recompile the game "ABC Expedition", which was mentioned earlier in section 5.1.1, with the compiler and memory management system developed in this thesis. Unfortunately, this is not possible within the time frame of this project and there are several good reasons for this. Firstly, the game is written in the full Standard ML language, including use of language features like functors, signatures and equality types, neither of which are currently supported by the CeXL programming language [CeXL10]. Secondly, the game requires bindings to the SDL (Simple Directmedia Layer) Library [SDL], which would take some effort to support in the implemented compiler. Thirdly, the compiler is currently neither optimizing enough for achieving interactive performance of a game, nor does it seem tuned enough to compile programs of that size in practice, not to mention bug-free and

complete enough to even compile realistic CeXL programs in general. Finally, the incremental part of garbage collection for reducing pause times was not completed on time and measuring pause times would be the most relevant part in relation to the statements in section 5.1.1. Hence, performance, memory usage and pause times will only be measured on the simpler programs in this thesis.

9.3 Results

The programs `fib.sml` and `tak.sml` cannot be executed without garbage collection, since my computer does not have anywhere near 80GB or 156GB of memory available.

By comparing the timings in figure 9.4 for the program `fib35.sml` when executed with or respectively without garbage collection, one sees that performing garbage collection in this case has a negative overhead. The higher performance when enabling garbage collection is due to not having to allocate much memory from the operating system and the associated much less required paging and much better cache behaviour. One also sees from figure 9.4 that allocating memory with the implemented method and performing garbage collection is around twice as fast as using `malloc` without `free` in this case. In fact, with garbage collection disabled, `malloc` is never faster. Thus, the allocation with this method, at least when allocating into entirely free regions, seems to be quite fast in comparison to `malloc`. Only when enabling garbage collection does `malloc` perform better on the `Count.sml` and the three `ExlSharing*.sml` programs. The overhead in these cases may come from both the collection and the allocation into recycled regions.

The program `msort.sml` keeps a larger amount of data live throughout the whole program execution, at least a list of 100,000 integers. Figure 9.5 shows that it requires 70BB of total system memory to execute the program, but this does not necessarily indicate how much live data there are, since it may just collect garbage too rarely. This program and the programs `ExlSharing*.sml` and `Trigon.sml` may be more indicative of a real program's memory behaviour than the previous ones, but this depends on the application.

Program	Time with `malloc`	Time w/o GC	User Time w/o GC	Time with GC	User Time with GC	Max. Collect Pause Time	Avg. Collect Pause Time	Max. Pause Time	Avg. Pause Time
`fib.sml`	N/A	N/A	N/A	382.27s	382.23s	45μs	3μs	56ms	0μs
`fib35.sml`	9.79s	5.70s	5.07s	5.41s	5.41s	6μs	2μs	1ms	0μs
`tak.sml`	N/A	N/A	N/A	656.69s	656.58s	18μs	5μs	74ms	0μs
`msort.sml`	1.95s	1.19s	1.09s	1.34s	1.30s	37ms	6ms	37ms	0μs
`Count.sml`	0.02s	0.01s	0.01s	0.21s	0.20s	12ms	5ms	12ms	0μs
`ExlSharing18.sml`	0.02s	0.01s	0.00s	0.42s	0.42s	19ms	11ms	19ms	1μs
`ExlSharing20.sml`	0.02s	0.01s	0.01s	1.06s	1.06s	37ms	29ms	36ms	3μs
`ExlSharing22.sml`	0.02s	0.01s	0.01s	3.61s	3.60s	110ms	100ms	108ms	11μs
`Trigons.sml`	2.61s	1.59s	1.32s	2.00s	1.74s	256ms	58ms	257ms	0μs

Figure 9.4: The *malloc*, no-collection and full-heap timings for the evaluated programs

When comparing the system memory usage for the program `fib35.sml` in figure 9.5, it can be seen that when using `malloc`, its overhead is so high that it uses almost twice as much memory as when allocating with the implemented method, in both cases without ever freeing any memory. The program

fib35.sml allocates small blocks of memory, at most 8 byte at a time[1], which suggests that malloc has an overhead of around 8 bytes per allocated memory area, which seems reasonable. malloc has almost as high space overhead for the program msort.sml, but slightly less in this case, since the program msort.sml sometimes allocates slightly larger chunks of memory, up to 16 bytes. For the programs Count.sml and ExlSharing*.sml, the space overhead of malloc is smaller, although these programs all allocate chunks of at most 12 bytes, but the average memory block size is 8 bytes in these cases, which further confirms that malloc must have around 8 bytes of overhead per allocated block.

Program	Sys. Mem. w. malloc	Sys. Mem. w/o GC	Sys. Mem. with GC	Allocated Memory	Number of Collections
fib.sml	N/A	N/A	3,096KB	80GB	228,085
fib35.sml	2,280MB	1,156MB	3,096KB	1,082MB	3,748
tak.sml	N/A	N/A	3,656KB	156GB	240,186
msort.sml	472MB	263MB	70MB	214MB	39
Count.sml	15MB	14MB	21MB	2,343KB	36
ExlSharing18.sml	15MB	14MB	21MB	2,344KB	36
ExlSharing20.sml	15MB	14MB	21MB	2,344KB	36
ExlSharing22.sml	15MB	14MB	21MB	2,344KB	36
Trigons.sml	818MB	549MB	554MB	312MB	8

Figure 9.5: The *malloc* and no-collection memory-usage as well as the full-heap memory-usage and number of collections for the evaluated programs

Programs like fib35.sml may not be very indicative of real-life program. The memory objects die very quickly and it should even be possible to compile this program such that only stack allocation is done, i.e. such that no heap allocation is ever done.

Looking at the results for the program Count.sml, it does not seem that creating 100,000 activation records and performing garbage collection on those present any problem. The stack-space usage is, however, not evaluated in these experiments, only the heap-space usage. Something similar can be said about the program Trigons.sml, where 1,000,000 list elements are created, traversed and retained throughout the program's running time.

It can be noticed that the programs Count.sml and ExlSharing*.sml have similar behaviour in both time and space usage, since they all contain the same counting loop, which accounts for most of the program's running time and allocation behaviour. There is only a slight difference in the amount of allocated memory, but it does not show in the kilobyte measurements in the presented tables. This is quite an ideal scenario for comparing the collection and pause times. The programs ExlSharing18.sml, ExlSharing20 and ExlSharing22 show the performance when garbage collecting a heap with varying amounts of exponential sharing of memory objects. It is quite evident that there is a problem with the time spent on memory management, since this is the only part of the computation which could take a significantly different amount of time between the Count.sml and the three ExlSharing*.sml programs. The time spent also clearly seems to be exponential in nature, which is as might be expected. Further, the pause times are directly affected by this, which is of course very unfortunate in a setting of memory management for interactive applications. I believe that by implementing the incremental stack tracing, as described but not yet implemented, the pause times could actually be reduced in this case, at least if using a time-based policy for the terminating the incremental stack-tracing.

Whether the exponential time-complexity of tracing exponentially shared heap data is a problem in

[1]The maximum and average size of allocated chunks of memory were also computed but will not be shown

practice would require more evaluation, since it is hard to tell from artificially created programs like these. It was argued in section 3.3 that many memory cells are short lived (point 1 in the list in section 3.3) and that few memory cells are shared (point 4 in the list). Several references from independent research were given for this. This would indicate that the time-complexity of the current implementation might not be a problem. It is for instance also well-known that Standard ML's type-inference algorithm takes exponential time in the worst case, but this normally never shows up in practical programs. I do not know whether or not something similar holds for the complexity of heap sharing. It can, however, be noticed that, even just while creating a list in a non-tail-recursive function, the sharing resulting from the persistent lists kept in the local stack-frames would give quadratic heap-sharing, which would currently have quadratic time-complexity to trace. This is actually also quite bad and may account for the collection times and pause times of the program `Count.sml`, in comparison to when not collecting garbage on that program (figure 9.4).

Looking at the right-most two columns in figure 9.4 with pause times including allocation times, they can be seen to quite consistently follow the pause times of the two previous columns, indicating that the largest pause times are currently the collection pauses, not allocation pauses. We therefore cannot say much about allocation pause-times. Cutting down these pause-times has not really been discussed anywhere, but one way of reducing them is to e.g. at most search only one recycled region for free lines per allocation and grab a new free region after that. Hence, the pause-times is reduced to the worst-case of searching one region with maximal fragmentation. Given the fixed and relatively small size of regions, this would likely be good enough.

Program	Time with GC	User Time with GC	Max. Collect Pause Time	Avg. Collect Pause Time	Sys. Mem. with GC Time	Allocated Memory Time	All Collections	Full Collections
`fib.sml`	443.23s	443.21s	13μs	2μs	6,360KB	80GB	80,587	4,747
`fib35.sml`	5.93s	5.93s	9μs	2μs	5,816KB	1,082MB	1108	62
`tak.sml`	736.95s	736.90s	87μs	4μs	9,096KB	156GB	96,117	5,649
`msort.sml`	1.27s	1.21s	13ms	1ms	101MB	214MB	39	1
`Count.sml`	0.03s	0.03s	12ms	123μs	19MB	2,343KB	166	1
`ExlSharing18.sml`	0.04s	0.04s	19ms	195μs	19MB	2,344	166	1
`ExlSharing20.sml`	0.08s	0.06s	37ms	441μs	19MB	2,344	166	1
`ExlSharing22.sml`	0.23s	0.22s	110ms	1ms	19MB	2,344	166	1
`Trigons.sml`	2.13s	1.83s	145ms	23ms	545MB	312MB	18	0

Figure 9.6: All measurements with in-place generational collection for the evaluated programs

Turning to the evaluation of the system with in-place generational collection, all measurements are shown in figure 9.6. There are both positive and negative effects to notice here.

Execution times are often higher in the generational system than the full-heap-collection system. This goes for the programs `fib.sml`, `fib35.sml`, `tak.sml` and `Trigons.sml`. The memory usage is also often higher in the generational system, evidenced by the programs `fib.sml`, `fib35.sml`, `tak.sml` and `msort.sml`. The generational system is also hardly ever able to reduce the amount of memory used, but this might be expected. The generational system does, however, not use prohibitively more time or memory than the full-heap-collection system in any of these cases.

For the programs `Count.sml` and `ExlSharing*.sml`, the execution time is quite significantly reduced by the generational system. These programs all have a large amount of live data. The reason for this

speed-up is that only one full-heap collection is performed for these programs, where the aforementioned problems with exponential heap-sharing seems to play a key role. If the collector only collects the most recently allocated memory objects, it does not collect the parts of the heap containing the shared objects. This will likely not hold in general, since if there are new references to the shared data, they will be traced again; or if the data with the sharing stays alive for a longer time, they will be traced for every full-heap collection. It can be noticed that the average collection pause times are not reduced in this case, as expected, since the one full-heap collection spends the time which gives rise to the pauses.

Looking at the number of collections performed, the generational system increases the number of collections in some cases (the `Count.sml`, `ExlSharing*.sml` and `Trigons.sml` programs) and decreases the number of collections in other cases (`fib.sml`, `fib35.sml` and `tak.sml`). The number of full-heap collections are always vastly reduced in the examples, but this is also expected and is the purpose of generational collection; so it seems to work.

All in all, the results seem promising, but require more work. In particular, cutting down pause-times has practically not been achieved. With the proposed but not yet fully implemented methods for incremental collection, this can hopefully be remedied.

Chapter 10

Future Work

The following is a list of suggested paths for future work:

- Several improvements could be made to the current compiler:
  - The current compiler does not generate particularly efficient code, nor does it work correctly without problems for the whole language. These issues should obviously be improved upon
  - The current compiler-implementation only stack-allocates local variables of simple types, never e.g. records (except for the empty record), tagged sum types and exceptions. It is very important to implement such stack allocation where possible, in order to reduce the pressure on the dynamic memory-management system
  - The exponential time spent on heap tracing is likely a problem, as mentioned in sections 8.4 and 9.3. Further investegations on this are important, possibly combined with investigations on using an object bitmap for marking, instead of only the currently used line-mark bitmaps
  - Reducing the required amount of stack space during heap tracing was indicated as being important in section 8.4, which makes it a very relevant consideration for future work
  - The extensions suggested in section 8.8 are obvious for future work
  - It might be worth-while to investigate the performance of heap-allocating activation records with the implemented memory-management system, as mentioned in section 3.2, since the method is supposed to give good locality of reference, in which case the allocation of stack and heap data simultaneously might result in good performance
- The only untyped compilation phase is the final phase of generating x86 assembly code from $FILM_L$ code and the only kind of problems not identified by the typed compilation phases are related to invalid memory access. For these reasons, I believe that it would be possible to extend the compiler to become a certifying compiler, without too much extra effort. The key challenge on this is precisely related to the memory management, not the final assembly code generation phase. The assembly code could become typed by using Typed Assembly Language (TAL) [Morr99]
- The compiler ML Kit [MLKit] uses regions in its memory management system and combines region inference with garbage collection. Since the implemented system also allocates into regions, which are easy to reclaim indepenently, the implemented garbage collection method would likely be suitable for incorporation into the ML Kit. Analogously, region inference would likely also be possible to incorporate into the implemented compiler, although implementing region inference seems to be a more difficult task than implementing the presented garbage-collection method

Chapter 11

Conclusion

This thesis has the following main contributions:

- Implementation of a modern type-specialized tracing of pointers on the stack, forming the root set for tracing heap-allocated memory
- Implementation of a modern memory-management method, based on recent research in [Blac08]
- Some initial measurements of pause times for the implemented memory-management method, which were not measured in the originally published research article [Blac08]
- Section 8.3 and some of the figures in the thesis gave a rough initial proposal for how one might let the programmer implement memory-management systems for compilers directly in the source language. The proposal would require more work, but is nevertheless an interesting prospect

I believe that in particular the combination of the first two contributions is novel, except that I am not sure what the trace specialization feature of MMTk does, mentioned in section 3.2.1 in [Blac08], which could potentially be similar. Also, I am not aware of the third contribution having been done anywhere else, which also makes this a novel contribution.

The results from this thesis are the following:

- The memory-management method implemented for this thesis seems to work fairly well in the current preliminary benchmark measurements, relating to performance in terms of time and space overhead. Keeping pause times low has not yet been achieved, which is important for using it with interactive applications. All other goals set out in section 1.3 have, in fact, been met
- The use of types seems to work well for the memory-management method, since it at least seems to eliminate some runtime conditional branches during the heap tracing, which may potentially be expensive, although the actual effect on performance of this has not been evaluated
- An large amount of code was developed during this project and the previous project that it builds on [Anoq10a] (to appear as [Anoq10b]). Many implementation mistakes were caught immediately up-front, by the type-checker of the compiler's implementation language, Standard ML, which is a type-safe language
- From the experiences gained by debugging the implemented compiler during the project, I found the type-checker for the typed intermediate languages $FILM_H$ and $FILM_L$ to be very useful and to catch several mistakes. Further, mistakes caught by the type-checker are usually reported

immediately after or very close to the problematic compilation phase, which greatly reduces the time to find the problem. I also found that, contrary to my expectations, those caught mistakes were almost never related to wrongly maintaining the types during the translation; it is surprisingly difficult to maintain types wrongly during translation, at least when the compiler itself is written in a type-safe language. I therefore highly recommend typed intermediate languages

Parts of this thesis were completed in a hurry in the last moment. I hope that the reader will bear with me this. An error-corrected version of this thesis will be made after the oral defence.

Acknowledgements

Thanks to Torben Ægidius Mogensen at the Department of Computer Science at the University of Copenhagen (DIKU) for supervising this project with constructive feedback and in particular for reading through parts of the thesis several times before the deadline, where many parts were written in great haste. Also thanks to Niels Hallenberg, Martin Elsman and Andrzej Filinski for a few short but very informative and useful conversations during this project. Finally thanks to Kathryn McKinley for answering questions regarding the Immix garbage collector and related methods.

Chapter 12

Appendices

All source code in this appendix (apart for the program `fib35.sml` which is available from elsewhere) is distributed under the GNU General Public Library License version 2 (LGPL) [LGPL2]. As a special exception, you are allowed to link it into programs without turning it into a separately loadable library, since this might severly harm performance in practice. You are otherwise required to follow the terms of the GNU General Public Library License version 2, such as releasing any changes made to the source code under a compatible license and making it as easily obtainable as any software within which it is used.

My future versions of the supplied code are not planned to be released under the same license and is supplied mostly in relation to this thesis.

12.1 The Program *fib35.sml*

```
val rec fib =
   fn 0 => 0
    | 1 => 1
    | n => fib (n - 1) + fib (n - 2)

structure Main =
   struct
      fun doit () =
 if 9227465 <> fib 35
    then raise Fail "bug"
 else ()
   end
```

12.2 The Function *transPrim* from the CeXL Compiler

The function `transPrim` in the compiler dispatches translation of primitive calls in the assembly-code generation for the $FILM_L$-language and may give some useful insights for the reader.

```
  (* Translate a primitive operation, which may result in either
     CPU-instructions or basis library calls *)
  fun transPrim (defVar, ty, (prim, args, handler, ann),
                 {align, label, insts}, blocks, regs, data) =
        if (String.isPrefix "___+" prim) orelse
           (* FIXME: Not entirely correct, since the offset has type int *)
           (String.isPrefix "___UnsafeMemoryAccess.addradd" prim) orelse
           (String.isPrefix "___Word.+" prim) orelse
```

```
   (String.isPrefix "___Int.+" prim) then
  transStdBinOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.IAdd, X86InstrRegAlloc.amongAll,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___-" prim) orelse
        (* FIXME: Not entirely correct, since the offset has type int *)
        (String.isPrefix "___UnsafeMemoryAccess.addrsub" prim) orelse
        (String.isPrefix "___Word.-" prim) orelse
        (String.isPrefix "___Int.-" prim)  then
  transStdBinOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.ISub, X86InstrRegAlloc.amongAll,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___~" prim) orelse
        (String.isPrefix "___Int.~" prim)  then
  (* There is room for optimization here... *)
  transStdBinOp
    (defVar, ty, prim,
     args@[FilmHlAbs.AConst {const = FilmHlAbs.CInt 0, ann = ann}],
     ann,
     X86AsmAbs.ISub, X86InstrRegAlloc.amongAll,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___Word.andb" prim) then
  transStdBinOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.IAnd, X86InstrRegAlloc.amongAll,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___Word.orb" prim) then
  transStdBinOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.IOr, X86InstrRegAlloc.amongAll,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___=" prim) orelse
        (String.isPrefix "___Word.=" prim) orelse
        (String.isPrefix "___Char.=" prim) orelse
        (String.isPrefix "___Int.=" prim)  then
  transStdRelOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.ICmp, X86AsmAbs.ISetz, false,
     X86InstrRegAlloc.amongFlagExtractable,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___<>" prim) orelse
        (String.isPrefix "___Word.<>" prim) orelse
        (String.isPrefix "___Char.<>" prim) orelse
        (String.isPrefix "___Int.<>" prim)  then
  transStdRelOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.ICmp, X86AsmAbs.ISetnz, false,
     X86InstrRegAlloc.amongFlagExtractable,
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___<" prim) orelse
        (String.isPrefix "___Word.<" prim) orelse
        (String.isPrefix "___Char.<" prim) orelse
        (String.isPrefix "___Int.<" prim)  then
  transStdRelOp
    (defVar, ty, prim, args, ann,
     X86AsmAbs.ICmp, X86AsmAbs.ISetl, false,
     X86InstrRegAlloc.amongFlagExtractable,
```

```
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___>" prim) orelse
         (String.isPrefix "___Word.>" prim) orelse
         (String.isPrefix "___Char.>" prim) orelse
         (String.isPrefix "___Int.>" prim)  then
   transStdRelOp
     (defVar, ty, prim, args, ann,
      X86AsmAbs.ICmp, X86AsmAbs.ISetl, true,
      X86InstrRegAlloc.amongFlagExtractable,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___<=" prim) orelse
         (String.isPrefix "___Word.<=" prim) orelse
         (String.isPrefix "___Char.<=" prim) orelse
         (String.isPrefix "___Int.<=" prim)  then
   transStdRelOp
     (defVar, ty, prim, args, ann,
      X86AsmAbs.ICmp, X86AsmAbs.ISetnl, true,
      X86InstrRegAlloc.amongFlagExtractable,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___>=" prim) orelse
         (String.isPrefix "___Word.>=" prim) orelse
         (String.isPrefix "___Char.>=" prim) orelse
         (String.isPrefix "___Int.>=" prim)  then
   transStdRelOp
     (defVar, ty, prim, args, ann,
      X86AsmAbs.ICmp, X86AsmAbs.ISetnl, false,
      X86InstrRegAlloc.amongFlagExtractable,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___*" prim) orelse
         (String.isPrefix "___Int.*" prim) orelse
         (* FIXME: Not handling unsigned multiply and overloading here *)
         (String.isPrefix "___Word.*" prim)  then
   transIMulOp
     (defVar, ty, prim, args, ann,
      X86InstrRegAlloc.amongAll,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___div" prim) orelse
         (String.isPrefix "___Int.div" prim) orelse
         (* FIXME: Not handling unsigned divide and overloading here *)
         (String.isPrefix "___Word.div" prim)  then
   transIDivOp
     (defVar, ty, prim, args, ann,
      X86InstrRegAlloc.amongAll,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 else if (String.isPrefix "___mod" prim) orelse
         (String.isPrefix "___Int.mod" prim) orelse
         (* FIXME: Not handling unsigned divide and overloading here *)
         (String.isPrefix "___Word.mod" prim)  then
   transIModOp
     (defVar, ty, prim, args, ann,
      X86InstrRegAlloc.amongAll,
      {align = align, label = label, insts = insts},
      blocks, regs, data)
 (* Now the memory primitives *)
 else if (String.isPrefix "___MrLineMarkAddr" prim) then
   (case args of
      [FilmHlAbs.AVar {name, ann = _}] =>
        let
          val commentedAnn =
                commentAnn
```

```
                ("let " ^ defVar ^ " = " ^ prim ^ "(" ^ name ^ ")")
                ann
          val insts' =
                FilmLMemoryPrimitiveGen.genLineMark
                  (name, commentedAnn, insts, regs)
        in
          ({align = align, label = label, insts = insts'},
           blocks, regs, data)
        end
    | _ =>
        Err.error
          (* FIXME: Allocate error code *)
          (0wx0101, "FFilmLToX86Asm.transPrim: Internal error: Expecting precisely one variable name argument for pri
  )
else if String.isPrefix "___MemAlloc" prim then
  transBasisCall
    (defVar, ty, ("___MemAlloc", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___MrClearLineMarks" prim then
  transBasisCall
    (defVar, ty, ("___MrClearLineMarks", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___MrReclaimGarbage" prim then
  transBasisCall
    (defVar, ty, ("___MrReclaimGarbage", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___MrShouldReclaimGarbage" prim then
  transBasisCall
    (defVar, ty, ("___MrShouldReclaimGarbage", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___MrFullOrPartialClearMask" prim then
  transBasisCall
    (defVar, ty, ("___MrFullOrPartialClearMask", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___MrShouldStillTrace" prim then
  transBasisCall
    (defVar, ty, ("___MrShouldStillTrace", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
(* Now for the other basis calls *)
else if String.isPrefix "___print" prim then
  transBasisCall
    (defVar, ty, ("___print", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___^" prim) orelse
        (String.isPrefix "___String.^" prim) then
  transBasisCall
    (defVar, ty, ("___String_append", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___CharVector_dimensions" prim) then
  transBasisCall
    (defVar, ty, ("___CharVector_dimensions", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___CharVector_empty" prim) then
  transBasisCall
    (defVar, ty, ("___CharVector_empty", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
```

```
else if (String.isPrefix "___CharVector_create" prim) then
  transBasisCall
    (defVar, ty, ("___CharVector_create", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___CharVector_sub" prim) then
  transBasisCall
    (defVar, ty, ("___CharVector_sub", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___CeXL_CharVector_update_" prim) then
  transBasisCall
    (defVar, ty, ("___CeXL_CharVector_update_", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___ord" prim) orelse
        (String.isPrefix "___Char_ord" prim) then
  (* FIXME: ___Char_ord and ___Char_chr should be translated as a
            FILM_L copy-expression, which does not generate any code *)
  transBasisCall
    (defVar, ty, ("___Char_ord", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if (String.isPrefix "___chr" prim) orelse
        (String.isPrefix "___Char_chr" prim) then
  transBasisCall
    (defVar, ty, ("___Char_chr", args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
(* A special kind of basis call: Calling a function pointer *)
else if String.isPrefix "___CallFunctionPtr" prim then
  transPtrCall
    (defVar, ty, (args, handler, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
(* Another special kind of basis call: Acquiring the stack base pointer *)
else if String.isPrefix "___GetStackBasePtrWithOffset" prim then
  transGetStackBasePtrWithOffset
    (defVar, ty, (args, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
(* Another special kind of basis call:
   Acquiring address of a variable, which is equivalent to
   adding or subtracting an offset to an address *)
else if (String.isPrefix "___GetVarAddrWithOffset" prim) then
  transGetVarAddrWithOffset
    (defVar, ty, (prim, false, args, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else if String.isPrefix "___GetVarAddr" prim then
  (* FIXME: Obsolete and not tested *)
  transGetVarAddr
    (defVar, ty, (args, ann),
     {align = align, label = label, insts = insts},
     blocks, regs, data)
else
  Err.error
    (* FIXME: Allocate error code *)
    (0wx0101, "FFilmLToX86Asm.transPrim: Internal error: Unknown primitive operation " ^ prim)
```

12.3 The Compiler-External Parts of the Memory-Management System in C

12.3.1 CBasisMarkRegionConfig.h: The Memory-Management Configuration

This file configures the compilation of the C-implementation. Code changes between the experiments were only made to this file. The compiler was the same binary without any external configuration in all experiments and generates exactly the same assembly code every time. The configuration shown is the default with in-place generational garbage collection, but no incremental garbage collection, since that was not completed in time.

```
#ifndef __CBASIS_MARK_REGION_CONFIG_H__
#define __CBASIS_MARK_REGION_CONFIG_H__

// This is an overriding garbage collection policy flag
//#define MRC_NEVER_GARBAGE_COLLECT                   1

// This is an overriding implementation flag.
// This requires the flag MRC_NEVER_GARBAGE_COLLECT
// to be set as well, to avoid disasters
//#define MRC_ALLOCATE_WITH_MALLOC                    1

// When this is enabled, allocation pause times
// are measured. This currently makes everything
// run _very_ slowly compared to not timing allocations
//#define MRC_TIME_ALLOCATION_PAUSES                  1

// When this flag is defined, in-place generational GC is performed
#define MRC_ENABLE_IN_PLACE_GENERATIONAL            1

// When this flag is defined, in-place generational GC
// triggers partial collections based on how many
// bytes have been allocated since the last GC
#define MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES 1

// When this flag is defined, incremental GC is enabled
// FIXME: This is buggy and incomplete
//#define MRC_ENABLE_INCREMENTAL                      1

// How long do we want an incremental garbage collect to run?
// FIXME: Lower values than 100ms seems to loop forever on ExlSharing22.sml,
//        but the idea is to get pause times down...
#define MRC_INCREMENTAL_TIMING_MICRO_SECONDS   100000

// Mark-region block-line size (currently not entirely configurable)
#define MRC_LINE_SIZE                             128

// Mark-region block size, 32 kilobytes (currently not entirely configurable)
#define MRC_BLOCK_SIZE                          32768

// Mark-region blocks to allocate at a time
#define MRC_BLOCKS_AT_A_TIME                       16


#endif // __CBASIS_MARK_REGION_CONFIG_H__
```

12.3.2 CBasisMain.c: The Main Program Interface

```
#include <stdio.h>
#include "CBasisMemStats.h"
#include "CBasisTiming.h"
#include <sys/time.h>
```

```
#include <sys/resource.h>
#include <unistd.h>

extern void ___asm_main(void);

int main(void)
{
    // Setup a stack size of 2GB, just to be sure...
    struct rlimit rl;
    rl.rlim_cur = 2000000000;
    rl.rlim_max = 2001000000;
    if(setrlimit(RLIMIT_STACK, &rl) != 0) {
puts("Stack limit not set, exiting to prevent crashes");
        return -1;
    }
    //rl.rlim_cur = 0;
    //rl.rlim_max = 0;
    //getrlimit(RLIMIT_STACK, &rl);
    //printf("Rlimit cur %d max %d\n", (int)rl.rlim_cur, (int)rl.rlim_max);

    BtiSetup();
    BtiResetTimeStats();
    BmsResetStats();
    ___asm_main();
    BmsShowStats();
    BtiShowTimeStats();

    return 0;
}
```

12.3.3 CBasisPrim.h

```
#include <stdio.h>

void ___print(char *str);

FILE **___TextIO_openOut(char *fileName);
void ___TextIO_closeOut(FILE **file);
void ___TextIO_output(FILE **file, int len, void *data);

void *___MemAlloc(int size);
void ___MemFree(void *addr);

// Clear all line marks, to initiate a full heap liveness trace
void ___MrClearLineMarks();

// The function that garbage collection is all about
void ___MrReclaimGarbage();

// Returns 1 if we should reclaim garbage, 0 otherwise
int ___MrShouldReclaimGarbage();

// Return the tracing-function clear-mask for either a full
// or a partial heap collection, depending on what it is time for
unsigned int ___MrFullOrPartialClearMask();

// Returns 1 if we should continue tracing back upwards the stack, 0 otherwise
int ___MrShouldStillTrace();
```

12.3.4 CBasisPrim.c: The Compiler Interface

This file defines all primitive-functions interfaced to the compiler.

```
#include "CBasisPrim.h"
```

```
#include "CBasisMarkRegionConfig.h"
#include "CBasisImmixBlockPool.h"
#include "CBasisMemStats.h"
#include "CBasisTiming.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

/* Print a Pascal-style string, suitable for CeXL and Standard ML */
void ___print(char *str) {
    unsigned int len = *((unsigned int *)str);
    /* 100 characters should be enough to hold two unsigned integers */
    char format[100];
    /* ...but we still use snprintf, not taking any chances for overflow */
    snprintf(format, 99, "%%%d.%ds", (int)len, (int)len);
    printf(format, str + 4);
}

char *___String_append(char *str1, char *str2) {
    unsigned int len1 = *((unsigned int *)str1);
    unsigned int len2 = *((unsigned int *)str2);

    char *result = ___MemAlloc(len1 + len2 + 4);

    *((unsigned int *)result) = len1 + len2;
    memcpy(result + 4, str1 + 4, len1);
    memcpy(result + 4 + len1, str2 + 4, len2);
    return result;
}

int ___CharVector_dimensions(char *str) {
    return *((unsigned int *)str);
}

char *___CharVector_empty() {
    char *result = ___MemAlloc(4);

    *((unsigned int *)result) = 0;
    return result;
}

char *___CharVector_create(int length, int value) {
    char *result = ___MemAlloc(4 + length);

    *((unsigned int *)result) = length;
    memset(result + 4, (char)value, length);

    return result;
}

int ___CharVector_sub(char *str, int index) {
    unsigned int len = *((unsigned int *)str);
    if(index >= 0 && index < len) {
        return (int)(*(str+4+index));
    } else {
puts("Index out of bounds in ___CharVector_sub");
        return 0;
    }
}

void ___CeXL_CharVector_update_(char *str, int index, int value) {
    unsigned int len = *((unsigned int *)str);
    if(index >= 0 && index < len) {
        *(str+4+index) = (char)value;
    } else {
```

```
        puts("Index out of bounds in ___CeXL_CharVector_update_");
    }
}

int ___Char_ord(int c) {
    // Well, this is trivial, since char is represented as 32-bit ints
    return c;
}

int ___Char_chr(int c) {
    // Well, this is trivial, since char is represented as 32-bit ints
    return c;
}

FILE **___TextIO_openOut(char *fileName) {
    FILE **file = malloc(sizeof(FILE *));
    *file = fopen(fileName, "w");

    return file;
}

void ___TextIO_closeOut(FILE **file) {
    fclose(*file);
    free(*file);
}

void ___TextIO_output(FILE **file, int len, void *data) {
    fwrite(data, 1, len, *file);
}

void *___MemAlloc(int size) {
#ifdef MRC_TIME_ALLOCATION_PAUSES
    BtiStartPauseTimer();
#endif
    BmsRegisterAllocatedBytes(size);
#ifdef MRC_ALLOCATE_WITH_MALLOC
    void *result = (void *)malloc(size);
#else
    void *result = (void *)IbpAlloc(size);
#endif
#ifdef MRC_TIME_ALLOCATION_PAUSES
    BtiReadPauseTimerMicro();
#endif
    return result;
}

void ___MemFree(void *addr) {
    free(addr);
}

// Clear all line marks, if initiating a full-heap liveness-trace
void ___MrClearLineMarks(unsigned int clearMask) {
    IbpClearLineMarkBitmapsInUse(clearMask);
}

// The function that garbage collection is all about
void ___MrReclaimGarbage() {
    IbpReclaimGarbage();
}

// Returns 1 if we should reclaim garbage, 0 otherwise
int ___MrShouldReclaimGarbage() {
    return IbpShouldReclaimGarbage();
}

// Return the tracing-function clear-mask for either a full
```

```
// or a partial heap collection, depending on what it is time for
unsigned int ___MrFullOrPartialClearMask() {
    return IbpFullOrPartialClearMask();
}

// Returns 1 if we should continue tracing back upwards the stack, 0 otherwise
int ___MrShouldStillTrace() {
#ifdef MRC_ENABLE_INCREMENTAL
    if(IbpShouldStillTrace()) {
//printf("+");
        return 1;
    } else {
//printf("-");
        return 0;
    }
    //return IbpShouldStillTrace();
#else
    return 1;
#endif
}
```

12.3.5 CBasisMemStats.h

```
#ifndef __CBASIS_MEM_STATS_H__
#define __CBASIS_MEM_STATS_H__

// This shows the operating system's view on memory usage (on Linux at least)
void BmsShowStats();

// Reset statistical information
void BmsResetStats();

// Functions for registering / gathering statistical information
void BmsRegisterAllocatedBytes(int bytes);
void BmsRegisterReclaimedBlock();
void BmsRegisterResetConservativeLive();
void BmsRegisterConservativeLiveBytes(int bytes);
void BmsRegisterCollectionComplete();
void BmsRegisterFullCollectionComplete();
void BmsRegisterPartialCollectionComplete();

// Functions for querying statistical information
int BmsGetConservativeLive();
int BmsGetPrevCollectionAllocBytes();
int BmsGetPartialPrevCollectionAllocBytes();
int BmsGetAllocBytes();

// Show gathered statistical information
void BmsShowStats();

#endif // __CBASIS_MEM_STATS_H__
```

12.3.6 CBasisMemStats.c

```
#include "CBasisMemStats.h"
#include <sys/types.h> // For pid_t (returned from getpid)
#include <unistd.h> // For getpid
#include <stdio.h>

// Some simple statistics on memory usage
int bmsAllocCnt = 0;
long long bmsAllocByteCnt = 0;
int bmsReclaimBlockCnt = 0;
int bmsAllocMaxBytes = 0;
int bmsCollectionCnt = 0;
```

```
int bmsFullCollectionCnt = 0;
int bmsPrevCollectionAlloc = 0;
int bmsPartialPrevCollectionAlloc = 0;
int bmsConservativeLive = 0;

// This shows the operating system's view on memory usage (on Linux at least)
// Notice: Using getrusage should have worked, but according to a post
//         on a website I have seen, that supposedly does not work on
//         e.g. Linux or Solaris (possibly other systems as well)
void BmsShowSysStats() {
    char buf[100];
    snprintf(buf, 99, "/proc/%u/statm", (unsigned)getpid());
    int sz = getpagesize();
    FILE* pf = fopen(buf, "r");
    if (pf) {
        unsigned size; //       total program size
        unsigned resident;//   resident set size
        unsigned share;//      shared pages
        unsigned text;//       code / executable segment
        unsigned lib;//        library (seems to be zero?)
        unsigned data;//       data/stack
        //unsigned dt;//         dirty pages (unused in Linux 2.6)
        fscanf(pf, "%u %u %u %u %u %u", &size, &resident, &share, &text, &lib, &data);
printf("Linux's view on memory usage, from %s:\n", buf);
printf("Used memory                   : %dkb / %dmb / %dgb\n",
       (int)((size * sz) / 1024),
       (int)((size * sz) / (1024 * 1024)),
       (int)((size * sz) / (1024 * 1024 * 1024)));
        printf("Resident set kilo bytes       : %d\n", (int)((resident * sz) / 1024));
printf("Shared kilo bytes             : %d\n", (int)((share * sz) / 1024));
printf("Code / executable segment kb  : %d\n", (int)((text * sz) / 1024));
//printf("Library                        : %d\n", (int)((lib * sz) / 1024));
        printf("Data segment size kilo bytes  : %d\n", (int)((data * sz) / 1024));
    }
    fclose(pf);
}

// Reset statistical information
void BmsResetStats() {
    bmsAllocCnt = 0;
    bmsAllocByteCnt = 0;
    bmsReclaimBlockCnt = 0;
    bmsAllocMaxBytes = 0;
    bmsCollectionCnt = 0;
    bmsFullCollectionCnt = 0;
    bmsPrevCollectionAlloc = 0;
    bmsPartialPrevCollectionAlloc = 0;
    bmsConservativeLive = 0;
}

void BmsRegisterAllocatedBytes(int bytes) {
    bmsAllocCnt++;
    bmsAllocByteCnt += bytes;
    if(bytes > bmsAllocMaxBytes) {
        bmsAllocMaxBytes = bytes;
    }
}

void BmsRegisterReclaimedBlock() {
    bmsReclaimBlockCnt++;
}

void BmsRegisterResetConservativeLive() {
    bmsConservativeLive = 0;
}
```

```
void BmsRegisterConservativeLiveBytes(int bytes) {
    bmsConservativeLive += bytes;
}

void BmsRegisterCollectionComplete() {
    bmsCollectionCnt++;
}

void BmsRegisterFullCollectionComplete() {
    bmsFullCollectionCnt++;
    bmsPrevCollectionAlloc = bmsAllocByteCnt;
    bmsPartialPrevCollectionAlloc = bmsAllocByteCnt;
}

void BmsRegisterPartialCollectionComplete() {
    bmsPartialPrevCollectionAlloc = bmsAllocByteCnt;
}

// Functions for querying statistical information
int BmsGetConservativeLive() {
    return bmsConservativeLive;
}

int BmsGetPrevCollectionAllocBytes() {
    return bmsPrevCollectionAlloc;
}

int BmsGetPartialPrevCollectionAllocBytes() {
    return bmsPartialPrevCollectionAlloc;
}

int BmsGetAllocBytes() {
    return bmsAllocByteCnt;
}


// Show gathered statistical information and the system's information
void BmsShowStats() {
    printf("Number of allocations         : %d\n", (int)bmsAllocCnt);
    printf("Allocated memory              : %dkb / %dmb / %dgb\n",
   (int)(bmsAllocByteCnt / 1024),
   (int)(bmsAllocByteCnt / (1024 * 1024)),
   (int)(bmsAllocByteCnt / (1024 * 1024 * 1024)));
    printf("Reclaimed regions             : %d\n", (int)bmsReclaimBlockCnt);
    printf("Largest memory block          : %d bytes\n", (int)bmsAllocMaxBytes);
    if(bmsAllocCnt > 0) {
        printf("Average memory block          : %d bytes\n", (int)(bmsAllocByteCnt / bmsAllocCnt));
    }
    printf("Number of collections         : %d\n", (int)bmsCollectionCnt);
    printf("Number of full collections    : %d\n", (int)bmsFullCollectionCnt);

    BmsShowSysStats();
}
```

12.3.7 CBasisTiming.h

```
#ifndef __CBASIS_TIMING_H__
#define __CBASIS_TIMING_H__

// Initialize the timer module
void BtiSetup();

void BtiResetTimeStats();
void BtiShowTimeStats();
```

```
void BtiStartPauseTimer();
long long BtiReadPauseTimerMicro();

// Start quick asynchronous timer in micro seconds
void BtiStartQuickAsyncTimerMicro(long microSeconds);

// A much quicker polling of whether the timer has fired than gettimeofday
int BtiIsQuickAsyncTimerFired();

#endif // __CBASIS_TIMING_H__
```

12.3.8 CBasisTiming.c

```
#include "CBasisTiming.h"
#include <stdio.h>
#include <stdlib.h>
#include <signal.h> // Used for (the timer_create family of functions?) and signal handling
#include <unistd.h>
#include <time.h> // Used for the clock_gettime family of functions (and the timer_create family of functions?)
#include <sys/time.h> // Used for gettimeofday

long long btiPauseTimer = 0;

int btiQuickAsyncTimerFired = 0;
timer_t btiQuickAsyncTimerId;
struct sigevent btiQuickAsyncTimerSev;
long long btiSlowIncrTimer = 0;

#define BTI_QUICK_ASYNC_TIMER_SIGNAL  SIGRTMIN

// Some simple timing statistics
long long btiMaxPauseTime = 0;
long long btiTotalPauseTime = 0;
long long btiPauseCnt = 0;

void BtiResetTimeStats() {
    btiMaxPauseTime = 0;
    btiTotalPauseTime = 0;
    btiPauseCnt = 0;
}

void BtiShowTimeStats() {
    printf("Max pause time               : %dus / %dms / %ds\n",
   (int)btiMaxPauseTime,
   (int)(btiMaxPauseTime / 1000),
   (int)(btiMaxPauseTime / 1000000));
    printf("Pauses                       : %d\n", (int)btiPauseCnt);
    if(btiPauseCnt > 0) {
printf("Avg pause time               : %dus / %dms / %ds\n",
       (int)(btiTotalPauseTime / btiPauseCnt),
       (int)((btiTotalPauseTime / btiPauseCnt) / 1000),
       (int)((btiTotalPauseTime / btiPauseCnt) / 1000000));
    }
}

long long BtiStartTimer() {
    // This works and is tested, but is even slower than gettimeofday
    // I believe that it is actually also less portable (not sure though)
    // However, I blieve that it should be more precise
    // (it is also nanoseconds, not microseconds)
    /*
    struct timespec timer;
    // CLOCK_REALTIME could also be used
    if(clock_gettime(CLOCK_MONOTONIC, &timer) != 0) {
        puts("Unable to get time for starting a timer, exiting");
```

```
        exit(-1);
    }
    return (((long long)timer.tv_nsec) >> 10) + ((long long)timer.tv_sec) * 1000000;
    */

    // This implementation with gettimeofday works and is tested,
    // but runs very slowly!
    struct timeval timer;
    if(gettimeofday(&timer, NULL) != 0) {
puts("Unable to get time for starting a timer, exiting");
        exit(-1);
    }
    return (long long)timer.tv_usec + ((long long)timer.tv_sec) * 1000000;
}

long long BtiReadTimerMicro(long long startedTimer) {
    // This works and is tested, but is even slower than gettimeofday.
    // I believe that it is actually also less portable (not sure though)
    // However, I blieve that it should be more precise
    // (it is also nanoseconds, not microseconds)
    /*
    struct timespec timer;
    // CLOCK_REALTIME could also be used
    if(clock_gettime(CLOCK_MONOTONIC, &timer) != 0) {
        puts("Unable to get time for reading a timer, exiting");
        exit(-1);
    }
    return (((long long)timer.tv_nsec) >> 10) + ((long long)timer.tv_sec) * 1000000 - startedTimer;
    */

    // This implementation with gettimeofday works and is tested,
    // but runs very slowly!
    struct timeval timer;
    if(gettimeofday(&timer, NULL) != 0) {
puts("Unable to get time for starting a timer, exiting");
        exit(-1);
    }
    return (long long)timer.tv_usec + ((long long)timer.tv_sec) * 1000000 - startedTimer;
}

void BtiStartPauseTimer() {
    btiPauseTimer = BtiStartTimer();
}

long long BtiReadPauseTimerMicro() {
    long long pauseTime = BtiReadTimerMicro(btiPauseTimer);

    if(pauseTime > btiMaxPauseTime) {
        btiMaxPauseTime = pauseTime;
    }
    btiTotalPauseTime += pauseTime;
    btiPauseCnt++;

    return pauseTime;
}

// A much quicker polling of whether the timer has fired than gettimeofday
int BtiIsQuickAsyncTimerFired() {
    // Alternative slow implementation which should work,
    // if using the precise clock_gettime implementation and it wasn't so slow!
    /*
    if(BtiReadTimerMicro(btiSlowIncrTimer) > 100000) {
return 1;
    } else {
return 0;
    }
```

```
    */
    // FIXME: Supposedly fast asynchronous implementation - possibly buggy
    return btiQuickAsyncTimerFired;
}

// A simple handler which just sets a flag, if the timer has fired
// FIXME: Supposedly fast asynchronous implementation - possibly buggy
static void BtiQuickAsyncTimerHandler(int sig, siginfo_t *si, void *uc) {
    btiQuickAsyncTimerFired = 1;
    signal(sig, SIG_IGN);
}

// Initialize the timer module
void BtiSetup() {
    struct sigaction sa;

    /* Establish handler for timer signal */

    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = BtiQuickAsyncTimerHandler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(BTI_QUICK_ASYNC_TIMER_SIGNAL, &sa, NULL) == -1) {
        puts("Error while trying to setup signal handler for timer, exiting");
exit(-1);
    }

    /* Create the timer */
    btiQuickAsyncTimerSev.sigev_notify = SIGEV_SIGNAL;
    btiQuickAsyncTimerSev.sigev_signo  = BTI_QUICK_ASYNC_TIMER_SIGNAL;
    btiQuickAsyncTimerSev.sigev_value.sival_ptr = &btiQuickAsyncTimerId;
    if (timer_create(CLOCK_REALTIME, &btiQuickAsyncTimerSev, &btiQuickAsyncTimerId) == -1) {
puts("Error creating asynchronous timer, exiting\n");
    }
}

// Start quick asynchronous timer in micro seconds
void BtiStartQuickAsyncTimerMicro(long microSeconds) {
    struct sigaction sa;
    struct itimerspec its;

    // Alternative slow implementation which should work,
    // if using the precise clock_gettime implementation and it wasn't so slow!
    //btiSlowIncrTimer = BtiStartTimer();

    // FIXME: Supposedly fast asynchronous implementation - possibly buggy
    /* Establish handler for timer signal */
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = BtiQuickAsyncTimerHandler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(BTI_QUICK_ASYNC_TIMER_SIGNAL, &sa, NULL) == -1) {
        puts("Error while trying to setup signal handler for timer, exiting");
exit(-1);
    }

    // Specify the timer
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = microSeconds * 1000;
    its.it_interval.tv_sec = its.it_value.tv_sec;
    its.it_interval.tv_nsec = its.it_value.tv_nsec;

    // Start the timer
    if (timer_settime(btiQuickAsyncTimerId, 0, &its, NULL) == -1) {
puts("Error starting asynchronous timer, exiting\n");
    }

    // Make sure that the timer fired flag is cleared
```

```
    btiQuickAsyncTimerFired = 0;
}
```

12.3.9 CBasisInternalHeapMng.h

```
#ifndef __CBASIS_INTERNAL_HEAP_MNG_H__
#define __CBASIS_INTERNAL_HEAP_MNG_H__

#include "CBasisMarkRegionConfig.h"

// Mark-region block size
#define IHM_BLOCK_SIZE   MRC_BLOCK_SIZE

// Allocate a given number of contiguous blocks, aligned to block size
void *IhmAllocAlignedBlocks(unsigned int blockCnt);

// Free a set of contiguous blocks allocated by IhmAllocAlignedBlocks
void IhmFreeAlignedBlocks(void *alignedAddr);

#endif // __CBASIS_INTERNAL_HEAP_MNG_H__
```

12.3.10 CBasisInternalHeapMng.c

```
#include "CBasisInternalHeapMng.h"
#include <fcntl.h>    // For O_RDONLY
#include <unistd.h>   // For getpagesize
#include <sys/mman.h> // For mmap and munmap
#include <stdio.h>
#include <stdlib.h>

// Compute an address aligned to the given size, which must be a power of two.
// The resulting address will always be after the given address,
// up to powTwoSize bytes after, in case it happened to be aligned
char *IhmAlignAddrToPowTwoSize(char *addr, unsigned int powTwoSize) {
  return ((char *)(((unsigned int)addr) & (~(powTwoSize - 1)))) + powTwoSize;
}

// Allocate a given number of contiguous blocks, aligned to block size
void *IhmAllocAlignedBlocks(unsigned int blockCnt) {
  // Allocate blockCnt blocks of memory, plus one extra block for alignment.
  // According to documentation, mmap aligns to page size, which is usually
  // 4096 bytes
  int fd = open("/dev/zero", O_RDONLY);
  //puts("Opened device");
  unsigned int bytes = IHM_BLOCK_SIZE * (blockCnt + 1);
  char *addr = mmap(NULL, bytes, PROT_WRITE | PROT_READ, MAP_PRIVATE, fd, 0);
  //printf("addr: %d\n", (int)addr);
  close(fd);

  if(!addr) {
      puts("System memory exhausted. Exiting to avoid crashes");
      exit(-1);
  }

  // Compute an adjusted address aligned to the block size.
  // It will always be after the given address, so there is always room before
  char *alignedAddr = IhmAlignAddrToPowTwoSize(addr, IHM_BLOCK_SIZE);
  //printf("alignedAddr: %d\n", (int)alignedAddr);

  if(((unsigned int)(alignedAddr - addr)) < 32) {
    puts("The allocated memory area was not aligned to at least 32 bytes, as expected. Exiting to avoid crashes");
    exit(-1);
  }

  // Use four bytes of the room before alignedAddr to remember
```

```
  // the allocated address and size, for the munmap later.
  // This part should scale to 64-bit memory management
  *((unsigned int *)(alignedAddr - 8)) = bytes;
  *((char **)(alignedAddr - 16)) = addr;

  return (void *)alignedAddr;
}

// Free a set of contiguous blocks allocated by IhmAllocAlignedBlocks
void IhmFreeAlignedBlocks(void *alignedAddr) {
  int bytes = *((unsigned int *)(((char *)alignedAddr) - 8));
  char *addr = *((char **)(((char *)alignedAddr) - 16));

  munmap(addr, bytes);
}

// When executing the code below on Gandalf
// (Ubuntu Linux 8.04 on Intel Pentium Duo Core 2, 2.4Ghz),
// It showed a page size of 4096.
/*
int main() {
  int sz = getpagesize();
  printf("Pagesize: %d\n", (int)sz);
}
*/
```

12.3.11 CBasisImmixBlockPool.h

```
#ifndef __CBASIS_IMMIX_BLOCK_POOL_H__
#define __CBASIS_IMMIX_BLOCK_POOL_H__

// Clear the line bitmap and the block list pointer
void IbpInitImmixBlock(void *block);

void *IbpGetImmixBlockAreaStart(void *block);

void *IbpGetImmixBlockAreaEnd(void *block);

void *IbpGetImmixBlockListPtr(void *block);

void IbpSetImmixBlockListPtr(void *block, void *ptr);

// This is like a Standard ML list cons, where blockPool is a pointer to
// the first block in the list and block is the new block to be cons'ed
void *IbpInsertImmixBlock(void *block, void *blockPool);

// Allocate initialize and put some blocks on the free list
void *IbpAllocFreshBlocks();

void *IbpGetFreeBlock();

// Find a block and an allocation span with at least bytes free bytes available
void IbpGetBlockWithFreeBytes(int bytes);

// currentPtr is about to pass endPtr, find a new allocation span
void IbpGetFreeAllocationSpan(int bytes);

// The final managed allocation function
void *IbpAlloc(int bytes);

// Clear all the line mark bitmaps of blocks which are in use
void IbpClearLineMarkBitmapsInUse(unsigned int clearMask);

// The function that garbage collection is all about
void IbpReclaimGarbage();
```

```
// Returns 1 if we should reclaim garbage, 0 otherwise
int IbpShouldReclaimGarbage();

// Return the tracing-function clear-mask for either a full
// or a partial heap collection, depending on what it is time for
unsigned int IbpFullOrPartialClearMask();

// Returns 1 if we should continue tracing back upwards the stack, 0 otherwise
int IbpShouldStillTrace();

// Mark a line bit corresponding to a given address
void IbpLineMarkAddr(void *addr);

#endif // __CBASIS_IMMIX_BLOCK_POOL_H__
```

12.3.12 CBasisImmixBlockPool.c: The Main Implementation

```
#include "CBasisInternalHeapMng.h"
#include "CBasisImmixBlockPool.h"
#include "CBasisMarkRegionConfig.h"
#include "CBasisMemStats.h"
#include "CBasisTiming.h"
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>

// Immix block line size
#define IBP_LINE_SIZE     MRC_LINE_SIZE

// Immix line bitmap size in bits
#define IBP_LINE_BITMAP_SIZE_BITS (IHM_BLOCK_SIZE / IBP_LINE_SIZE)

// Immix line bitmap size in bytes
#define IBP_LINE_BITMAP_SIZE (IBP_LINE_BITMAP_SIZE_BITS / 8)

// Immix blocks to allocate at a time
#define IBP_BLOCKS_AT_A_TIME MRC_BLOCKS_AT_A_TIME

#define IBP_BLOCK_PAYLOAD_SIZE (IHM_BLOCK_SIZE - IBP_LINE_SIZE)

// The clear mask when performing a full heap collection
#define IBP_CLEAR_MASK_FULL_COLLECTION         ((unsigned int)~0x1)

// The clear mask when performing only a partial heap collection
#define IBP_CLEAR_MASK_PARTIAL_COLLECTION      ((unsigned int)~0x0)

// The global lists of allocated blocks,
// classified as unavailable, recycled or free
void *ibpUnavailableBlockPool = NULL;
void *ibpRecycledBlockPool = NULL;
void *ibpFreeBlockPool = NULL;

// Bump pointer allocation
char *currentPtr = NULL;
char *endPtr = NULL;

// Memory of the previously initiated (i.e. currently executing) collection
unsigned int ibpInitiatedClearMask = IBP_CLEAR_MASK_FULL_COLLECTION;
int ibpCompletedLastCollection = 1;

void IbpInitImmixBlock(void *block) {
    // Clear the line bitmap and the block list pointer
    memset(block, 0, IBP_LINE_BITMAP_SIZE + sizeof(void *));
}
```

```
void IbpClearMarksImmixBlock(void *block) {
    unsigned int *lineBitmap = (unsigned int *)block;
    //printf("Clearing block at %d\n", (int)block);
    // Reset 256 bits (32768 / 128 lines), taking up 256 / 32 = 8 32-bit words
    *(lineBitmap + 0) = 0;
    *(lineBitmap + 1) = 0;
    *(lineBitmap + 2) = 0;
    *(lineBitmap + 3) = 0;
    *(lineBitmap + 4) = 0;
    *(lineBitmap + 5) = 0;
    *(lineBitmap + 6) = 0;
    *(lineBitmap + 7) = 0;
}

// Returns 1 if block is in use (i.e. at least one bit is marked), 0 otherwise
int IbpHasMarksImmixBlock(void *block) {
    unsigned int *lineBitmap = (unsigned int *)block;
    unsigned int commonBitmap =
*(lineBitmap + 0) |
*(lineBitmap + 1) |
*(lineBitmap + 2) |
*(lineBitmap + 3) |
*(lineBitmap + 4) |
*(lineBitmap + 5) |
*(lineBitmap + 6) |
*(lineBitmap + 7);

    // This makes the returned value compatible with the compiler's bool true
    return (int)(commonBitmap != 0);
}

// Returns 1 if block is full (i.e. all bits are marked), 0 otherwise
int IbpIsMarkedFullImmixBlock(void *block) {
    unsigned int *lineBitmap = (unsigned int *)block;
    unsigned int commonBitmap =
*(lineBitmap + 0) &
*(lineBitmap + 1) &
*(lineBitmap + 2) &
*(lineBitmap + 3) &
*(lineBitmap + 4) &
*(lineBitmap + 5) &
*(lineBitmap + 6) &
*(lineBitmap + 7);

    // We OR with 1 because the first line in the bitmap contains header
    // information and is therefore never used, and because we can never
    // be sure to use the first line marked in the first bit of the
    // following 7 words in the line mark bitmap, due to conservative marking.
    // This makes the returned value compatible with the compiler's bool true
    // The constant is 32^2 - 1
    return (int)((commonBitmap | 1) == (int)(-1));
}

// Show the line marks of a given block
void IbpShowImmixBlockLineMarks(void *block) {
    int word = 0;
    int bitNumber = 0;
    unsigned int bitMask  = (((unsigned int)0x1) << bitNumber);
    unsigned int *lineMarkWord = (unsigned int *)block;
    while((char *)lineMarkWord < (((char *)block) + IBP_LINE_BITMAP_SIZE)) {
        while(bitNumber < 32) {
            bitMask = (unsigned int)(0x1 << bitNumber);
    if((*lineMarkWord & bitMask) == 0) {
                printf(".");
    } else {
```

```
                printf("1");
    }

            // Advance to next bit
    bitNumber++;
}

// Advance to next 32-bit word (4 bytes) and start at first two bits
        bitMask = 0x1;
        bitNumber = 0;
lineMarkWord++;
word++;
printf("|\n");
    }
}

void *IbpGetImmixBlockAreaStart(void *block) {
    return (void *)(((char *)block) + IBP_LINE_SIZE);
}

void *IbpGetImmixBlockAreaEnd(void *block) {
    return (void *)(((char *)block) + IHM_BLOCK_SIZE);
}

void *IbpGetImmixBlockListPtr(void *block) {
    return *((void **)(((char *)block) + IBP_LINE_BITMAP_SIZE));
}

void IbpSetImmixBlockListPtr(void *block, void *ptr) {
    *((void **)(((char *)block) + IBP_LINE_BITMAP_SIZE)) = ptr;
}

void *IbpInsertImmixBlock(void *block, void *blockPool) {
    // This is like a Standard ML list cons, where blockPool is a pointer to
    // the first block in the list and block is the new block to be cons'ed
    IbpSetImmixBlockListPtr(block, blockPool);
    return block;
}

// Allocate initialize and put some blocks on the free list
void *IbpAllocFreshBlocks() {
    //puts("Allocating fresh blocks");
    char *blocks = IhmAllocAlignedBlocks(IBP_BLOCKS_AT_A_TIME);
    void *block = NULL;
    int cnt;
    for(cnt = 0; cnt < IBP_BLOCKS_AT_A_TIME; cnt++) {
// Put the blocks onto the free list, such that they are
        // listed avaiable in consecutive allocation order
block = (void *)(blocks + (IBP_BLOCKS_AT_A_TIME - cnt - 1) * IHM_BLOCK_SIZE);
IbpInitImmixBlock(block);
//printf("IbpAllocFreshBloks: IbpInsertImmixBlock(%d, %d)\n", (int)block, (int)ibpFreeBlockPool);
ibpFreeBlockPool = IbpInsertImmixBlock(block, ibpFreeBlockPool);
    }

    // Check that things went as we expect
    if(blocks != ibpFreeBlockPool) {
        puts("Expecting the address of the allocated blocks to be the same as the final free list address");
    }

    return ibpFreeBlockPool;
}

void *IbpGetFreeBlock() {
    void *result;

    //puts("IbpGetFreeBlock");
```

```
    if(!ibpFreeBlockPool) {
        // There are no free blocks in the free pool, allocate some
IbpAllocFreshBlocks();
    }

    // There are blocks in the free pool, get one
    result = ibpFreeBlockPool;
    //printf("Got free block at address %d\n", (int)result);

    // Remove block from free pool
    ibpFreeBlockPool = IbpGetImmixBlockListPtr(result);
    // Add block to unvailable pool
    //printf("IbpGetFreeBlock: IbpInsertImmixBlock(%d, %d)\n", (int)result, (int)ibpUnavailableBlockPool);
    ibpUnavailableBlockPool = IbpInsertImmixBlock(result, ibpUnavailableBlockPool);

    //puts("Inserted block");
    // Initialize bump allocation pointers
    currentPtr = IbpGetImmixBlockAreaStart(result);
    endPtr = IbpGetImmixBlockAreaEnd(result);

    // Clear the memory area, for
    // debugging if anything important is overwritten
    //memset(currentPtr, 0, IBP_BLOCK_PAYLOAD_SIZE - IBP_LINE_SIZE);

    //puts("Cleared block");


    return result;
}

// Find an allocation span with at least bytes free bytes available
void IbpGetLinesWithFreeBytes(void *block, void *startPtr, int bytes) {
    //printf("IbpGetLinesWithFreeBytes %d\n", (int)bytes);
    if((unsigned int)(startPtr - block) >= IHM_BLOCK_SIZE) {
puts("Something completely wrong with the parameters for IbpGetLinesWithFreeBytes");
        exit(-1);
    }
    unsigned int word = ((unsigned int)(startPtr - block)) >> (5 + 7);
    // Bit number is the first of two bits to test, due to conservative line marking
    unsigned int bitNumber = ((((unsigned int)startPtr) >> 7) & 0x1f);
    if(bitNumber == 0) {
bitNumber = 1;
    }
    unsigned int bitsMask  = (((unsigned int)0x1) << bitNumber);
    unsigned int *lineMarkWord = ((unsigned int *)block) + word;
    while((char *)lineMarkWord < (((char *)block) + IBP_LINE_BITMAP_SIZE)) {
        // Continue searching for a bit until the end of the word
        //printf("Searching word %d\n", (int)word);
while(bitNumber < 32) {
            //printf("Searching bit %d\n", (int)bitNumber);
    if((*lineMarkWord & bitsMask) == 0) {
                // A bit is free, which
// is a potential start address for allocation
currentPtr =
    ((char *)block) +
    (word * 32 + bitNumber) * IBP_LINE_SIZE;
if(currentPtr < (char *)startPtr) {
    puts("currentPtr < startPtr");
                    exit(-1);
}
if((currentPtr < (char *)block) ||
   currentPtr > ((char *)block + IHM_BLOCK_SIZE)) {
    puts("currentPtr out of block");
                    exit(-1);
}
```

```
// Now continue to find an end address
// Notice: This is usable as a quick and dirty test:
        // endPtr = currentPtr + IBP_LINE_SIZE;

// Advance to next bit
        bitNumber++;
bitsMask = bitsMask << 1;

                // Continue searching for a marked bit until the end of the block
while((char *)lineMarkWord < (((char *)block) + IBP_LINE_BITMAP_SIZE)) {
                    // Continue searching for a bit until the end of the word
    while(bitNumber < 32) {
                        //printf("Searching bit %d\n", (int)bitNumber);
                     if((*lineMarkWord & bitsMask) != 0) {
    // We encountered a dirty bit, compute the span
    // we have found and check if it is good enough
                            endPtr =
                        ((char *)block) +
        (word * 32 + bitNumber) * IBP_LINE_SIZE;

    // Found an allocation span
                            // Check if the area found is big enough
            if((unsigned int)(endPtr - currentPtr) < bytes) {
                // We found an area which is not big enough,
                // continue searching recursively
                                IbpGetFreeAllocationSpan(bytes);
            }

                            return;
}

                        // Advance to next bit
                bitNumber++;
                        bitsMask = bitsMask << 1;
    }

                    // Advance to next 32-bit word (4 bytes) and start at second bit
                    bitsMask = 0x1;
                    bitNumber = 0;
            lineMarkWord++;
                    word++;
}

// We have now searched the entire block for a dirty end bit,
// but found none, so we can allocate into the rest of
                // the block. endPtr is thus the end of the block
endPtr = ((char *)block) + IHM_BLOCK_SIZE;

// Check if the area found is big enough
if((unsigned int)(endPtr - currentPtr) < bytes) {
    // We found an area which is not big enough,
    // continue searching recursively
                    IbpGetFreeAllocationSpan(bytes);
}

// Clear the memory area, for
// debugging if anything important is overwritten
//memset(currentPtr, 0, IBP_LINE_SIZE);

//printf("Found bit %d in block %d, currentPtr %d, endPtr %d\n", (int)bitNumber, (int)block, (int)currentPtr, (int)endPtr);

return;
    }

            // Advance to next bit
    bitNumber++;
```

```
            bitsMask = bitsMask << 1;
}

// Advance to next 32-bit word (4 bytes) and start at second bit
        bitsMask = 0x2;
        bitNumber = 1;
lineMarkWord++;
        word++;
    }

    // Exhausted the search in this block, get a new block and search there
    IbpGetBlockWithFreeBytes(bytes);
}

// Find a block and an allocation span with at least bytes free bytes available
void IbpGetBlockWithFreeBytes(int bytes) {
    void *recycledBlock;

    //printf("IbpGetBlockWithFreeBytes %d\n", (int)bytes);
    if(ibpRecycledBlockPool) {
        //puts("ibpRecycledBlockPool is a pointer (not NULL)");
// There are blocks in the recycled pool, get one
recycledBlock = ibpRecycledBlockPool;
//printf("Got recycled block at address %d\n", (int)recycledBlock);

// Remove block from recycled pool
ibpRecycledBlockPool = IbpGetImmixBlockListPtr(recycledBlock);

// Add block to unvailable pool
//printf("IbpGetBlockWithFreeBytes: IbpInsertImmixBlock(%d, %d)\n", (int)recycledBlock, (int)ibpUnavailableBlockPool);
ibpUnavailableBlockPool = IbpInsertImmixBlock(recycledBlock, ibpUnavailableBlockPool);

// Initialize bump allocation pointers. When giving it the
// start of the block as initial search pointer, it will
        // skip the initial header line, due to conservative line marking
        IbpGetLinesWithFreeBytes(recycledBlock, IbpGetImmixBlockAreaStart(recycledBlock), bytes);
    } else {
        //puts("ibpRecycledBlockPool is NULL");
// There are no more recycled blocks, get a free block
        IbpGetFreeBlock();
    }
    //puts("Got block with free bytes");
}

// currentPtr is about to pass endPtr, find a new allocation span
void IbpGetFreeAllocationSpan(int bytes) {
    //printf("IbpGetFreeAllocationSpan %d\n", (int)bytes);
    if((((unsigned int)endPtr) & (IHM_BLOCK_SIZE - 1)) == 0) {
        //printf("endPtr is aligned with block, at %d\n", (int)endPtr);
// endPtr is aligned with a block, so this is the end of a block.
// Find a new block, either recycled, if there are any, or a free block
        IbpGetBlockWithFreeBytes(bytes);
    } else {
//printf("endPtr is not aligned with block, at %d\n", (int)endPtr);
// endPtr is not aligned with a block, find next free hole of
// lines within the current block
IbpGetLinesWithFreeBytes((unsigned int *)(((unsigned int)endPtr) & (~(IHM_BLOCK_SIZE - 1))), endPtr, bytes);
    }
    //puts("Got free allocation span");
}

// The final managed allocation function
void *IbpAlloc(int bytes) {
    void *result;

    //printf("IbpAlloc(%d)\n", (int)bytes);
```

```
    // We don't want to allocate zero bytes, ever
    // FIXME: Change the translation instead to avoid this
    if(bytes == 0) {
        bytes = 4;
    }

    // FIXME: Handle allocation of large objects too...
    if((unsigned int)(endPtr - currentPtr) >= bytes) {
        // We can directly allocate the desired amount of memmory
result = currentPtr;
currentPtr += bytes;
return result;
    } else {
        //printf("We need more memory because %d - %d = %d < %d\n", (int)endPtr, (int)currentPtr, (int)(endPtr - currentPtr)
// We need more memory
// FIXME: Check if it is time to trigger a collection...
// Acquire memory
        IbpGetFreeAllocationSpan(bytes);
//IbpGetFreeBlock();

if((unsigned int)(endPtr - currentPtr) < bytes) {
    printf("Unable to allocate %d bytes of memory. Exiting to avoid crashes\n", (int)bytes);
            exit(-1);
}

        // Bump allocate the block of memory
result = currentPtr;
currentPtr += bytes;
        return result;
    }
}

// Clear line mark bitmaps in a linked list of mark-region blocks
void IbpClearLineMarkBitmapsInList(void *blockList) {
    char *blockPtr = (char *)blockList;

    while(blockPtr != NULL) {
        //printf("IbpClearMarksImmixBlock(%d)\n", (int)blockPtr);
IbpClearMarksImmixBlock(blockPtr);
        blockPtr = IbpGetImmixBlockListPtr(blockPtr);
    }
}

// Clear all the line mark bitmaps of blocks which are in use
void IbpClearLineMarkBitmapsInUse(unsigned int clearMask) {
    //printf("IbpClearLineMarkBitmapsInList(%d)\n", (int)ibpUnavailableBlockPool);
    BtiStartPauseTimer();

#ifdef MRC_ENABLE_INCREMENTAL
    // Start the timer for incremental garbage collection termination
    //printf("\n(%d)S", (int)clearMask);
    BtiStartQuickAsyncTimerMicro(MRC_INCREMENTAL_TIMING_MICRO_SECONDS);
    if(ibpCompletedLastCollection) {
    ibpCompletedLastCollection = 0;
#endif

#ifdef MRC_ENABLE_IN_PLACE_GENERATIONAL
    if(clearMask == IBP_CLEAR_MASK_FULL_COLLECTION) {
        // Line-mark bitmaps should only be cleared on full-heap allocations
IbpClearLineMarkBitmapsInList(ibpUnavailableBlockPool);
    }
#else
    // When in-place generational GC is enabled, collections are
    // always full-heap collections, so all mark bits are always cleared
    IbpClearLineMarkBitmapsInList(ibpUnavailableBlockPool);
#endif
```

```
#ifdef MRC_ENABLE_INCREMENTAL
    }
#endif // MRC_ENABLE_INCREMENTAL
    //puts("Cleared");
    // We should always make sure that there are no recycled blocks,
    // when initiating a full GC, which is the only time where
    // the line mark bitmaps are fully cleared with this function
    //IbpClearLineMarkBitmapsInList(ibpRecycledBlockPool);
}

// The function that garbage collection is all about
void IbpReclaimGarbage() {
    char *newUnavailableBlocks = (char *)NULL;
    char *blockPtr = (char *)ibpUnavailableBlockPool;
    char *nextBlockPtr = (char *)NULL;

#ifdef MRC_ENABLE_INCREMENTAL
    if(!BtiIsQuickAsyncTimerFired()) {
        // The incremental tracing has completed, so it is safe to reclaim garbage
#endif // MRC_ENABLE_INCREMENTAL

    //puts("Reclaiming garbage");
    if(ibpInitiatedClearMask == IBP_CLEAR_MASK_FULL_COLLECTION) {
// We only reset the conservative liveness estimate when
        // initiating full-heap collections
BmsRegisterResetConservativeLive();
    }

    while(blockPtr != NULL) {
        //printf("Line mark bitmap of block at %d\n", (int)blockPtr);
        //IbpShowImmixBlockLineMarks(blockPtr);

        // Remember pointer to next block in list, before it is overwritten
nextBlockPtr = IbpGetImmixBlockListPtr(blockPtr);

if(IbpHasMarksImmixBlock(blockPtr)) {
    // The block is still in use, check if it is recyclable
    if(IbpIsMarkedFullImmixBlock(blockPtr)) {
                // The block is full, retain it in the unavailable list
        //printf("Keeping block at %d\n", (int)blockPtr);
        newUnavailableBlocks =
            IbpInsertImmixBlock(blockPtr, newUnavailableBlocks);

        BmsRegisterConservativeLiveBytes(IBP_BLOCK_PAYLOAD_SIZE);
    } else {
// The block is only partially full, recycle it

// Check if we are already allocating into the recycled block
if((currentPtr >= (char *)IbpGetImmixBlockAreaStart(blockPtr)) &&
   (currentPtr <= (char *)IbpGetImmixBlockAreaEnd(blockPtr))) {
    // We are already allocating into this block, which is
    // actually perfect. Just continue doing that
    // and mark it as unavailable, rather than recycled
    newUnavailableBlocks =
        IbpInsertImmixBlock(blockPtr, newUnavailableBlocks);
} else {
    ibpRecycledBlockPool =
IbpInsertImmixBlock(blockPtr, ibpRecycledBlockPool);
}

                // This estimate is very conservative...
BmsRegisterConservativeLiveBytes(IBP_BLOCK_PAYLOAD_SIZE);
    }
} else {
            // Block is unused, reclaim it
            //printf("Reclaiming block at %d\n", (int)blockPtr);
```

```
    ibpFreeBlockPool = IbpInsertImmixBlock(blockPtr, ibpFreeBlockPool);
    // Clear the memory area, for
    // debugging if anything important is overwritten
    //memset(IbpGetImmixBlockAreaStart(blockPtr), 0,
    //      IBP_BLOCK_PAYLOAD_SIZE - IBP_LINE_SIZE);

    // Check that we were not allocating into the now reclaimed block,
    // since it would not be valid to continue with that when it
            // is marked as free
    if((currentPtr >= (char *)IbpGetImmixBlockAreaStart(blockPtr)) &&
       (currentPtr <= (char *)IbpGetImmixBlockAreaEnd(blockPtr))) {
// We were allocating into this block, reset bump allocation
currentPtr = NULL;
                endPtr = NULL;
    }
    BmsRegisterReclaimedBlock();
}

        // Iterate to next block in list, by using the saved next pointer
blockPtr = nextBlockPtr;
    }

    // An updated list of used blocks has now been constructed
    ibpUnavailableBlockPool = newUnavailableBlocks;

    // Make sure that bump allocation pointers are properly re-initialized
    if(currentPtr == NULL) {
// Get a free block to allocate into
// FIXME: Try to get a recycled block instead...
IbpGetFreeBlock();
    }

    //puts("Reclaimed garbage");
    BmsRegisterCollectionComplete();
    if(ibpInitiatedClearMask == IBP_CLEAR_MASK_FULL_COLLECTION) {
BmsRegisterFullCollectionComplete();
    } else {
BmsRegisterPartialCollectionComplete();
    }
#ifdef MRC_ENABLE_INCREMENTAL
        //printf("C");
        ibpCompletedLastCollection = 1;
    } // End of incremental GC completed
    //else {
    //    printf("I");
    //}
#endif // MRC_ENABLE_INCREMENTAL
    // We have to measure pause times on incremental and completed collections
    BtiReadPauseTimerMicro();
}

// Returns 1 if we should reclaim garbage, 0 otherwise
int IbpShouldReclaimGarbage() {
    int conservativeLive = BmsGetConservativeLive();
    int prevAllocBytes = BmsGetPrevCollectionAllocBytes();
    int allocBytes = BmsGetAllocBytes();

    // Compute the newly allocated number of bytes since the last full collection
    int newlyAllocBytes = allocBytes - prevAllocBytes;

    // Compute how much we estimate has been allocated in the heap
    int heapEstimate = 0;

    // Make sure that we at least consider one block as being live always
    if(conservativeLive <= 0) {
        conservativeLive = IBP_BLOCK_PAYLOAD_SIZE;
```

```
    }

    // Compute how much we estimate has been allocated in the heap
    heapEstimate = conservativeLive + newlyAllocBytes;

#ifdef MRC_NEVER_GARBAGE_COLLECT
    return 0;
#else
    // We want to reclaim garbage, if the heap is estimated to have allocated
    // 3 times as much memory as was conservatively live on previous collection
    int shouldReclaimFullHeap =
((conservativeLive * 3) < heapEstimate) &&
(ibpRecycledBlockPool == NULL);

#ifdef MRC_ENABLE_IN_PLACE_GENERATIONAL
#ifdef MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES
    // Compute the newly allocated number of bytes
    int prevPartialAllocBytes = BmsGetPartialPrevCollectionAllocBytes();
    int newlyPartialAllocBytes = allocBytes - prevPartialAllocBytes;

    // Return true if we should do either a full-heap or a partial collection
    return shouldReclaimFullHeap ||
// This partial heap collection policy is to do a partial collection,
        // if we have allocated more than previously conservatively live
        (newlyPartialAllocBytes * 3 > conservativeLive * 1);// > MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES);
#endif // MRC_ENABLE_IN_PLACE_GENERATIONAL
#else
    // Only check for full-heap collections
    return shouldReclaimFullHeap;
#endif // MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES
#endif // MRC_NEVER_GARBAGE_COLLECT
}

// Return the tracing-function clear-mask for either a full
// or a partial heap collection, depending on what it is time for
unsigned int IbpFullOrPartialClearMask() {
    // Notice: This is ugly cut and paste, but at least the
    // IbpShouldReclaimGarbage _should_ be efficient and not
    // call any functions, since it is called _very_ frequently.
    // This function may be less efficient, as it is called only
    // when actually initiating collections
    int conservativeLive = BmsGetConservativeLive();
    int prevAllocBytes = BmsGetPrevCollectionAllocBytes();
    int allocBytes = BmsGetAllocBytes();

    // Compute the newly allocated number of bytes
    int newlyAllocBytes = allocBytes - prevAllocBytes;

    // Compute how much we estimate has been allocated in the heap
    int heapEstimate = 0;

    // Make sure that we at least consider one block as being live always
    if(conservativeLive <= 0) {
        conservativeLive = IBP_BLOCK_PAYLOAD_SIZE;
    }

    // Compute how much we estimate has been allocated in the heap
    heapEstimate = conservativeLive + newlyAllocBytes;

    //printf("conservativeLive: %d, newlyAllocBytes: %d, heapEstimate: %d\n",
    //       (int)conservativeLive, (int)newlyAllocBytes, (int)heapEstimate);

    // We want to reclaim the full heap, if the heap is estimated to
    // have allocated 3 times as much memory as was conservatively
    // live on previous collection
    int shouldReclaimFullHeap =
```

```
((conservativeLive * 3) < heapEstimate) &&
(ibpRecycledBlockPool == NULL);

#ifdef MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES
#ifdef MRC_ENABLE_INCREMENTAL
    if(ibpCompletedLastCollection) {
//ibpCompletedLastCollection = 0;
#endif // MRC_ENABLE_INCREMENTAL
if(shouldReclaimFullHeap) {
            ibpInitiatedClearMask = IBP_CLEAR_MASK_FULL_COLLECTION;
        } else {
            ibpInitiatedClearMask = IBP_CLEAR_MASK_PARTIAL_COLLECTION;
}
#ifdef MRC_ENABLE_INCREMENTAL
    } else {
        ibpInitiatedClearMask = IBP_CLEAR_MASK_PARTIAL_COLLECTION;
    }
#endif // MRC_ENABLE_INCREMENTAL
#else
#ifdef MRC_ENABLE_INCREMENTAL
    // Always do full-heap collections if partial collection is disabled
    if(ibpCompletedLastCollection) {
        //ibpCompletedLastCollection = 0;
#endif // MRC_ENABLE_INCREMENTAL
ibpInitiatedClearMask = IBP_CLEAR_MASK_FULL_COLLECTION;
#ifdef MRC_ENABLE_INCREMENTAL
    } else {
        ibpInitiatedClearMask = IBP_CLEAR_MASK_PARTIAL_COLLECTION;
    }
#endif // MRC_ENABLE_INCREMENTAL
#endif // MRC_IN_PLACE_GENERATIONAL_PARTIAL_WORK_BASED_BYTES

    return ibpInitiatedClearMask;
}

// Returns 1 if we should continue tracing back upwards the stack, 0 otherwise
int IbpShouldStillTrace() {
    return !BtiIsQuickAsyncTimerFired();
}


// Mark a line bit corresponding to a given address
void IbpLineMarkAddr(void *addr) {
    unsigned int *lineMarkBitmapAddr = (unsigned int *)(((unsigned int)addr) & (~(IHM_BLOCK_SIZE - 1)));
    unsigned int word = ((unsigned int)(((unsigned int)addr) & (IHM_BLOCK_SIZE - 1))) >> (5 + 7);
    unsigned int *lineMarkWordAddr = (unsigned int *)(lineMarkBitmapAddr + word);
    // FIXME: Or shift from max unsigned int towards the right?
    unsigned int bit  = (((unsigned int)0x3) << ((((unsigned int)addr) >> 7) & 0x1f));

    // Mark the line corresponding to addr
    *lineMarkWordAddr = (*lineMarkWordAddr) | bit;
}
```

12.3.13 Makefile: Compiling and Linking Programs

```
# -Wall enables all warning messages. -lc and -lm link with libc and libm
Compile:
gcc -g -Wall -fstack-check CBasisTiming.c CBasisMemStats.c CBasisInternalHeapMng.c CBasisImmixBlockPool.c CBasisPrim.c genasm.s CBasisMain.c -lc -lm -lrt -o program

# Unused
#CBasis.s:
# gcc -S CBasisPrim.c CBasisMain.c
```

12.3.14 Test/TestCBasisImmixBlockPool.c: Educative Standalone Unit-Test

```
#include "../CBasisImmixBlockPool.h"
```

```
#include "../CBasisMemStats.h"
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *addr;
    int *mem[1000];
    int size = 4;
    int cnt;

    for(cnt = 0; cnt < 1000; cnt++) {
        mem[cnt] = IbpAlloc(size);
        *(mem[cnt]) = size;
size += 4;
        printf("%d, ", (int)(mem[cnt]));
    }
    puts("");

    size = 4;
    for(cnt = 0; cnt < 1000; cnt++) {
if(size != *(mem[cnt])) {
    printf("Expecting size %d to be stored, but found size %d\n",
   (int)size, (int)(*(mem[cnt])));
            exit(-1);
}
size += 4;
    }

    puts("Successful allocation");
    BmsShowStats();

    // Write at a few select memory addresses
    addr = ((mem[30]) + 20);
    *addr = 42;
    IbpLineMarkAddr(addr);
    addr = ((mem[40]) + 24);
    *addr = 50;
    IbpLineMarkAddr(addr);
    addr = ((mem[50]) + 28);
    *addr = 58;
    IbpLineMarkAddr(addr);

    // Reclaim memory
    IbpReclaimGarbage();

    BmsShowStats();

    for(cnt = 0; cnt < 32000; cnt++) {
        addr = IbpAlloc(4);
        *addr = 0;
    }

    // Verify that the memory addresses are retained
    if(*((mem[30]) + 20) != 42) {
puts("Expecting value 42 to be retained in memory");
        exit(-1);
    }
    if(*((mem[40]) + 24) != 50) {
puts("Expecting value 50 to be retained in memory");
        exit(-1);
    }
    if(*((mem[50]) + 28) != 58) {
puts("Expecting value 58 to be retained in memory");
        exit(-1);
    }
```

```
    BmsShowStats();

    // Reclaim memory
    IbpReclaimGarbage();

    puts("Success");
    BmsShowStats();

    exit(0);
}
```

12.3.15 Test/Makefile: Running the Unit-Test

Compile the unit-test with `make clean` followed by `make test2`.

```
testAll: test1 test2 test3


test1:
gcc ../CBasisInternalHeapMng.c TestCBasisInternalHeapMng.c -o test1
./test1

test2:
gcc ../CBasisTiming.c ../CBasisMemStats.c ../CBasisInternalHeapMng.c ../CBasisImmixBlockPool.c TestCBasisImmixBlockPool.c -lrt -o test2
./test2

test3:
gcc ../CBasisTiming.c ../CBasisMemStats.c ../CBasisInternalHeapMng.c ../CBasisImmixBlockPool.c TestCBasisImmixBlockPoolMark.c -lrt -o test3
./test3

clean:
rm -f ./test1
rm -f ./test2
rm -f ./test3
```

Bibliography

[AMLB09] Online reference for the Ánoq SML Basis Library (0.8.2), by Ánoq of the Sun, 2009-05-18: `http://www.hardcoreprocessing.com/pro/anoqsmlbasis/`

[Anoq04] Ánoq of the Sun. *Comparison of Typed Memory Management via Static Capabilities with Type Preserving Garbage Collectors*. Department of Computer Science at the University of Copenhagen (DIKU), handed in as part of the course *Types and Programming Languages* by Fritz Henglein and Andrzej Filinski, 2004.

[Anoq10a] Ánoq of the Sun. *Advanced Compiler Middle and Back-Ends*. Department of Computer Science at the University of Copenhagen (DIKU) and Hardcore Processing. March 24th, 2010.

[Anoq10b] Ánoq of the Sun. *Advanced Compiler Middle and Back-Ends*. Hardcore Processing. To appear at: `http://www.cex3d.net/cexl/`

[AhoS86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley 1986.

[AhoL07] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools (Second Edition)*. Pearson Education 2007.

[Appe88] Andrew W. Appel, John R. Ellis and Kai Li. "Real-Time Concurrent Collection on Stock Multiprocessors". 1988.

[Appe89] Andrew W. Appel. "Simple Generational Garbage Collection and Fast Allocation". In *Sofware and Practice and Experience, 19(2)*, pages 171-183. 1989.

[Appe98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[Auer07] Joshua Auerbach, Perry Cheng, David Grove, David F. Bacon, Michael Dawson, Darren Hart, Bob Blainey, Mike Fulton and Mark Stoodley. "Design and Implementation of a Comprehensive Real-Time Java Virtual Machine". In *EMSOFT'07*. 2007.

[Azat03] H. Azatchi and E. Petrank. "Integrating Generations with Advanced Reference Counting Garbage Collectors". In *International Conference of Compiler Contruction*, Warsaw, Poland. April 2003.

[Baco01] David F. Bacon and V. T. Rajan. "Concurrent Cycle Detection in Reference Counting Systems". In *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001, Budapest, Hungary, June 18-22*, pages 207-235. Springer-Verlag, 2001.

[Baco03] David F. Bacon, Perry Cheng and V. T. Rajan. "A Real-Time Garbage Collector with Low Overhead and Consistent Utilization". In *Principles of Programming Languages 2003*. ACM, 2003.

[Baco04] David F. Bacon, Perry Cheng and David Grove. "Garbage Collection for Embedded Systems". In *Proceedings of Fourth ACM International Conference On Embedded Software (EMSOFT)*. 2004.

[Bake78] H. G. Baker. "List Processing in Real-Time on a Serial Computer". In *Commun. ACM 21, 4 (April 1978)*, pages 280-294. 1978.

[Bake85] Brenda Baker, E.G. Coffman and D.E. Willard. "Algorithms for Resolving Conflicts in Dynamic Storage Allocation". In *Journal of the ACM, 32(2)*, pages 327-343. April 1985.

[Bake92] Henry G. Baker. "The Treadmill: Real-Time Garbage Collection without Motion Sickness". In *ACM SIGPLAN Notices 1992*. 1992.

[BenY02] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper and Victor Leikehman. "An Algorithm for Parallel Incremental Compaction". In *Proceedings of the Third International Symposium on Memory Management (ISMM'02)*. June 2002.

[Blac03a] Stephen M. Blackburn, Perry Cheng and Kathryn S. McKinley. "A Garbage Collection Bakeoff in a Java Memory Management Toolkit (JMTk)". Technical Report TR-CS-03-02, ANU. March 2003.

[Blac03b] Stephen M. Blackburn and Kathryn S. McKinley. "Ulterior Reference Counting: Fast Garbage Collection without a Long Wait". In *OOPSLA 2003 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. ACM, 2003.

[Blac04a] Stephen M. Blackburn. "Oil and Water? High Performance Garbage Collection in Java with MMTk". In *ICSE 2004, 26th International Conference on Software Engineering*. 2004.

[Blac04b] Stephen M. Blackburn and Antony L. Hosking. "Barriers: Friend or Foe?". In *ISMM'04*. October 2004.

[Blac08] Stephen M. Blackburn and Kathryn S. McKinley. "Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance". In *PLDI'08*. ACM, June, 2008.

[Boeh88] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment". In *Software Practice and Experience, 18(9)*, pages 807-820. 1988.

[Boeh91] Hans-Juergen Boehm, Alan J. Demers and Scott Shenker. "Mostly Parallel Garbage Collection". In *ACM SIGPLAN Notices, 26(5)*, pages 157-164. 1991.

[Boya03] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., Martin Rinard. "Ownership Types for Safe Region-Based Memory Management in Real-Time Java". In *Programming Language Design and Implementation*. 2003.

[Bren89] R. P. Brent. "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation". In *ACM Transactions in Programming Languages and Systems, 11(3)*, pages 388-403. July 1989.

[Broo84] R. A. Brooks. "Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware". In *Proceedings of the ACM Conference on Lisp and Functional Programming, 1984*, pages 256-262. 1984.

[CeXL10] Ánoq of the Sun. *Design and Definition of CeXL and ξ-Calculus Version 0.9.3*. Hardcore Processing, March 2010. `http://www.cex3d.net/cexl/definition/`

[Cors03] Angelo Corsaro and Ron K. Cytron. "Efficient Memory-Reference Checks for Real-time Java". In *LCTES 2003*. 2003.

[Chea00] A. M. Cheadle, A. J. Field, S. Marlow, S. L. Peyton Jones and R. L. While. "Non-Stop Haskell". In *International Conference on Functional Programming, 2000*, pages 257-76. 2000.

[Chea04] A. M. Cheadle and A. J. Field. "Exploring the Barrier to Entry - Incremental Generational Garbage Collection for Haskell". In *Int. Symp. on Memory Management 2004*. 2004.

[Chen01] P. Cheng and G. Blelloch. "A Parallel, Real-Time Garbage Collector". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices, 33, 6*, pages 125-136. May 2001.

[Chen06] Wen-ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao and Weihaw Chuang. "Profile-Guided Proactive Garbage Collection for Locality Optimization". In *Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation 2006*. 2006.

[Chri84] T. W. Christopher. "Reference Count Garbage Collection". In *Software Practice and Experience, 14(6)*, pages 503-507. June 1984.

[Clic05] C. Click, G. Tene and M. Wolf. "The Pauseless GC Algorithm". In *VEE'05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (2005)*, pages 46-56. 2005.

[Demm90] A. Demmers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow and S. Shenker. "Combining Generational and Conservative Garabage Collection: Framework and Implementations". In *ACM Symposium on the Pinciples of Programming Languages*, pages 261-269. ACM, 1990.

[Dhur03] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, Chris Lattner. "Memory Safety Without Runtime Checks or Garbage Collection". In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems*. 2003.

[Dhur05] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, Chris Lattner. "Memory Safety without Garbage Collection for Embedded Applications". In *ACM Transactions on Embedded Computing Systems*. 2005.

[Dijk78] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten and E. F. M. Steffens. "On-the-Fly Garbage Collection: An Exercise in Cooperation". In *Communications of the ACM, 21(11)*, pages 965-975. November 1987.

[Doli94] Damien Doligez and Georges Gonthier. "Portable, Unobtrusive Garbage Collection for Multiprocessor Systems". In *Principles of Programming Languages (POPL)*. 1994.

[Doli93] Damien Doligez and Xavier Leroy. "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML". In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, ACM SIGPLAN Notices*, pages 113-123. USENEX Association, 1993.

[Pizl10] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton and Jan Vitek. "Schism: Fragmentation-Tolerant Real-Time Garbage Collection". In *PLDI'10*. June 2010.

[Fram07] D. Framption, D. F. Bacon, P. Cheng and D. Grove. "Generational Real-Time Garbage Collection: A Three-Part Invention for Young Objects". In *European Conference on Object-Oriented Programming*, pages 101-125. July, 2007.

[GHC] Online reference for the Glasgow Haskell Compiler (GHC), by Simon Peyton Jones et. al. `http://www.haskell.org/ghc/`

[GNUStep] Online reference for the Open Source project GNUStep: `http://www.gnustep.org` and `http://wiki.gnustep.org`

[Hall10] Niels Hallenberg. Private communication. 2010.

[Hall02] Niels Hallenberg, Martin Elsman and Mads Tofte. "Combining Region Inference and Garbage Collection". In *PLDI'02*. ACM, June 2002.

[Hall99] Niels Hallenberg. *Combining Garbage Collection and Region Inference in the ML Kit*. Master's thesis, Department of Computer Science, University of Copenhagen (DIKU), 1999.

[HcPE01] Online reference for the Hardcore Processing entertainment pages, currently containing five games from 2000 and 2001 with sound and graphics by Virtual Effects and Fantasies and programming by Ánoq of the Sun at Hardcore Processing. `http://www.hardcoreprocessing.com/entertainment`

[Henn93] Wade Hennessey. "Real-Time Garbage Collecion in a Multimedia Programming Language". In *OOPSLA-gc*. 1993.

[John92] Ralph E. Johnson. "Reducing the Latency of a Real-Time Garbage Collector". In *Letters on Programming Languages and Systems, 1(1)*, pages 46-58. March 1992.

[Jone96] Richard Jones and Rafael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management* (reprint from 2007). John Wiley and Sons, 1996.

[Knut97] Donald Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1997.

[Lang87] Bernard Lang and Francis Dupont. "Incremental Incrementally Compacting Garbage Collection". In *SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, volume 22(7) of ACM SIGPLAN Notices*, pages 253-263. ACM Press, 1987.

[Leib83] H. Lieberman and C. E. Hewitt. "A Real-Time Garbage Collector Based on the Lifetimes of Objects". In *Communications of the ACM, 26(6)*, pages 419-429. 1983.

[LGPL2] Online reference for the GNU General Public Library License version 2, by Richard Stallman and the Free Software Foundation: `http://www.gnu.org/licenses/old-licenses/lgpl-2.0.html`

[Mann05] Tobias Mann, Morgan Deters, Rob Legrand and Ron K. Cytron. "Static Determination of Allocation Rates to Support Real-Time Garbage Collection". In *ACM SIGPLAN Notices 2005*. 2005.

[MLKit] Online reference for the *ML Kit* compiler, by Mads Tofte, Martin Elsman et. al. `http://www.it-c.dk/research/mlkit/index.php/Main_Page`

[MLton] Online reference for the *MLton* compiler, by Stephen Weeks et. al. `http://www.mlton.org/`

[MLto07] Online reference for the MLton compiler, by Stephen Weeks et al, version 20070826: `http://www.mlton.org`

[Moge09] Torben Æ. Mogensen. *Basics of Compiler Design*. Diku, 2009. Freely available online from: `http://www.diku.dk/hjemmesider/ansatte/torbenm/Basics/`

[Morr99] Greg Morrisett, David Walker, Karl Crary and Neal Glew. "From System F to Typed Assembly Language". In *ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 21, Issue 3*, pages 527-568. ACM, May 1999.

[Much97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press 1997.

[Nett93] S. Nettles and J. O'Toole. "Real-Time Garbage Collection". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, SIGPLAN Notices, 28, 6*, pages 217-226. June 1993.

[Nort87] S. C. North and J. H. Reppy. "Concurrent Garbage Collection on Stock Hardware". In *Functional Programming Languages and Computer Architecture (Portland, Oregon, September 1987), vol. 274 of Lecture Notes in Computer Science*, pages 113-133. 1987.

[SDL] Online reference for the *SDL* (Simple Directmedia Layer) Library, by Sam Lantinga et. al. `http://www.libsdl.org`

[Shao95] Zhong Shao and Andrew W. Appel. *A Type-Based Compiler for Standard ML*. PRINCETON-CS-TR-487-95. March 28, 1995.

[SML97] Robin Milner, Mads Tofte, Robert Harper, David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[SMLB04] Online reference for the Standard ML Basis Library (2004 edition), by John Reppy et al: `http://www.standardml.org/Basis/`

[SMLNJ] Online reference for the *Standard ML of New Jersey* compiler, by David MacQueen et. al. `http://www.smlnj.org/`

[Spoo05] D. Spoonhower, G. Blelloch and R. Harper. "Using Page Residency to Balance Tradeoffs in Tracing Garbage Collection". In *ACM International Conference on Virtual Execution Environments*. ACM, June 2005.

[Stee75] Guy L. Steele. "Multiprocessing Compactifying Garbage Collection". In *Communications of the ACM, 18(9)*, pages 495-508. September 1975.

[Stef99] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. Ph.D. thesis, University of Massachusetts, 1999.

[Toft97] Mads Tofte and Jean-Pierre Talpin. "Region-Based Memory Management". In *Information and Computation, 132(2)*, pages 109-176. 1997.

[Toft94] Mads Tofte and Jean-Pierre Talpin. "Implementation of the Typed Call-by-Value λ-Calculus Using a Stack of Regions". In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, ACM SIGPLAN Notices*, pages 188-201. ACM Press, January 1994.

[Tolm94] Andrew Tolmach. *Tag-Free Garbage Collection Using Explicit Type Parameters*. Portland State University, 1994.

[Unga84] D. M. Ungar. "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm". In *ACM SIGPLAN Notices, 19(5)*, pages 157-167. April 1984.

[Vech06] Martin T. Vechev. "Correctness-Preserving Derivation of Concurrent Garbage Collection Algorithms". ACM Press 2006.

[Walk00] David Walker, Karl Crary and Greg Morrisett. "Typed Memory Management via Static Capabilities". In *ACM Programming Languages and Systems*, pages 701-771. 2000.

[Wang00] Daniel C. Wang and Andrew W. Appel. *Type-Preserving Garbage Collectors (Extended Version)*, Tech Report TR-624-00. Department of Computer Science, Princeton University, 2000.

[Wang02] Daniel Chen-An Wang. *Managing Memory with Types*. Ph.D. thesis, Department of Computer Science, Princeton University, January 2002.

[Wils92] Paul R. Wilson, Michael S. Lam and Thomas G. Moher. "Caching Considerations for Generational Garbage Collection". In *LFP, 1992*, pages 32-42. 1992.

[Wils93] Paul R. Wilson and Mark S. Johnstone. "Truly Real-Time Non-Copying Garbage Collection". In *OOPSLA-gc*. 1993.

[Wils95] Paul R. Wilson. "ReL Real-Time GC (was Re: Widespread C++ Competency Gap)". *USENET comp.lang.c++*. January 1995.

[Whit90] Jon L. White. "Three Issues in Object-Oriented Garbage Collection". In *OOPSLA-gc*. 1990.

[Yuas90] T. Yuasa. "Real-Time Garbage Collection on General-Purpose Machines". In *Journal of Systems and Software 11(3) (March 1990)*, pages 181-198. March 1990.

[Zorn90] B. Zorn. *Barrier Methods for Garbage Collection*. Tech. Report CU-CS-494-90. University of Colorado at Boulder, 1990.