Warsaw University of Technology

Faculty of Mathematics and Information Science

Master thesis
Computer science

Applications of memory management optimization techniques in script languages interpreters

Zastosowanie optymalizacji pamięci w interpreterach języków skryptowych

Author:
Julian Zubek

Supervisor:
Krzysztof Kaczmarski Ph.D.

Warsaw, September 2012

Supervisor's signature

Author's signature

Abstract

The goal of this work was to analyse and compare the strategies of automatic memory management used in the interpreters of popular script languages. The methods for optimisation of memory management were also searched for. Three popular script languages were analysed: Perl, Python and Ruby. The main memory management mechanisms used in the official implementations of these lanaguages were identified. In the case of Perl this was reference counting, in the case of Ruby this was the mark-and-sweep garbage collector, and in the case of Python the hybrid approach mixing both of these techniques were used.

The next step was to find what fraction of the script's execution time is spent on memory management. Experiments showed that for all three languages and simple test programs the value was between 20% and 50%. This much time spent on garbage collection is the result of treating all kinds of values—even simple numerics—as objects. The results justify the need for solutions that would optimise existing memory management mechanisms.

A short survey of memory optimisations proposed in the literature was carried out. I have searched for mechanisms supplementary to standard memory management techniques. I have paid special attention to the concept of stack allocation, which allows to free all allocated objects at once when the function exits. Another concept that was described more broadly was compile time garbage collection.

Selected mechanisms were implemented as extensions of Skarb—an experimental Ruby to C compiler developed by the author of this thesis as a part of bachelor thesis. The goal of the bachelor thesis was to provide basic compiler functionality. In this thesis mechanisms for stack allocation and memory reuse as well as a fairly sophisticated static code analysis mechanism based on a connection graph were introduced.

The connection graph provides information about relations between objects and references. It allows to perform reachability analysis and identify function local objects. It also makes simple object liveness analysis possible. Connection graph proved to be a convenient and universal form of code abstraction, suitable for implementing selected optimizations.

The results obtained from the experimental tests showed that in the case of some recursive functions even 60% of the objects created during program execution can be stack-allocated. The performance gain introduced by stack allocation in such cases reached 40%. The mechanism of local object reuse allowed to achieve 15% speedups in tests containing operations on long arrays executed in a loop.

Streszczenie

Celem pracy była analiza i porównanie metod automatycznego zarządzania pamięcią stosowanych w interpreterach języków skryptowych oraz poszukiwanie możliwości ich optymalizacji. Analizie zostały poddane trzy popularne języki obiektowe: Perl, Python oraz Ruby. Zidentyfikowano główne mechanizmy zarządzania pamięcią stosowane w oficjalnych implementacjach tych języków. W przypadku Perla było to zliczanie referencji, w przypadku Ruby'ego *garbage collector* typu *mark and sweep*, w przypadku Pythona rozwiązanie hybrydowe wykorzystujące oba te mechanizmy.

Kolejnym krokiem było zbadanie jak duża część czasu wykonania programu napisanego w danym języku jest poświęcana na zarządzanie pamięcią. Eksperymenty wykazały, że dla wszystkich trzech języków i prostych programów testowych wartość ta mieści się w przedziale 20% do 50%. Tak duże wielkości wynikają z konsekwentnego traktowania w badanych językach wszystkich wartości – nawet prostych typów liczbowych – jako pełnoprawnych obiektów. Wyniki te uzasadniają potrzebę poszukiwania rozwiązań przyśpieszających obecne mechanizmy zarządzania pamięcią.

Dokonano przeglądu literatury pod kątem rozwiązań wspomagających tradycyjne techniki zarządzania pamięcią. Szczególną uwagę poświęcono koncepcji alokacji na stosie, która pozwala na błyskawiczne zwolnienie zaalokowanych obiektów przy powrocie z funkcji. Przyjrzano się też z zainteresowaniem opisywanym rozwiązaniom statycznego odśmiecania pamięci podczas kompilacji (*compile time garbage collection*).

Wybrane mechanizmy zaimplementowano jako rozszerzenia eksperymentalnego kompilatora Ruby'ego Skarb, tworzonego przez autora tej pracy w ramach pracy inżynierskiej. Celem pracy inżynierskiej było zapewnienie podstawowej funkcjonalności kompilatora, natomiast w ramach tej pracy dodana została możliwość alokacji obiektów na stosie, wielokrotnego wykorzystywania raz zaalokowanej pamięci oraz zaawansowany mechanizm statycznej analizy kodu oparty na grafie połączeń (*connection graph*).

Graf połączeń zawiera informacje o relacjach pomiędzy występującymi w programie obiektami a referencjami. Pozwala on na przeprowadzenia analizy dostępności obiektów w konkretnych funkcjach programu i identyfikacje obiektów lokalnych względem funkcji, oraz na prostą analizę czasu życia obiektów. Graf połączeń okazał się bardzo wygodną i uniwersalną formą abstrakcji kodu, pozwalającą na zrealizowanie założonych optymalizacji.

Testy wykazały, że w przypadku niektórych funkcji rekurencyjnych ponad 60% wszystkich obiektów tworzonych podczas wykonania programu może być zaalokowanych na stosie. Uzyskane dzięki temu przyśpieszenie czasu wykonania programu sięga 40%. Mechanizm statycznego odzyskiwania pamięci pozwolił na uzyskanie przyśpieszenia rzędu 15% w testach zawierających operacje na długich tablicach wykonywane w pętli.

Contents

Introduction

Script languages such as Perl, Python and Ruby have gained popularity among programmers. They are no longer regarded as toy languages and it is becoming more and more common to use them to develop complex applications. This trend puts a demand on fast and optimised interpreters and compilers. In my bachelor thesis [30], written together with Jan Stępień, I proposed a way to meet this demand by translating the Ruby source code to the C source code and then bt using one of the optimised C compilers. This approach resulted in serious speedups in comparison with the original Ruby implementation, but the performance gain was often limited by the performance of the garbage collector. This observation led me to studies of memory management techniques in the context of script languages. In this work I analyse techniques used in the interpreters of popular languages and propose methods of optimisation. Selected optimisations were implemented and tested experimentally.

The rest of this thesis is structured as follows: Chapter 1 shortly characterises the languages which are being taken into account and describes the memory management strategies used in their interpreters. It also presents some of the results of experimental studies showing how a large portion of program's execution time is spent on garbage collection. In Chapter 2 various ways of optimising memory management are mentioned. Two techniques, i.e., stack allocation and compile time garbage collection, are more fully described. Chapter 3 contains the details of implementation of those techniques and describes the connection graph—a form of code abstraction used for necessary analyses. Chapter 4 presents the obtained experimental results and Chapter 5 is a short conclusion of the thesis.

Chapter 1

Script languages and memory management

1.1 Script languages

Script language (or scripting language) is a term which is commonly used by programmers but lacks a formal definition. Traditionally, it refers to a programming language designed to extend the functionality of or to facilitate other programs rather than to create standalone applications. Common use cases of script languages include:

- automation of job execution and control (e.g., in an operating system shell),
- text processing,
- writing configuration files,
- creating plugins,
- building dynamic web pages (with scripts executed both on the client side and the server side).

Nowadays, however, the distinction between script languages and general purpose languages is becoming blurry. It is not that uncommon to implement stand-alone applications in script languages. Programmers often claim that this allows to focus on high-level program logic without low-level details obscuring the view. Many script languages have a reputation of being very flexible, thus of speeding up the development process, allowing fast prototyping and making testing easier. Because of this the term "script language" in modern usage often refers to a certain set of language features rather than to its purpose.

Most of the script languages are interpreted languages executed by a dedicated interpreter or some kind of virtual machine. Scripts (files containing a code written in a script language) can usually be executed directly without compilation beforehand. Script languages rarely require that a programmer declare the variables before usage; they are also often dynamically typed, thus

allowing values of different types to be assigned to the same variable. Many script languages are called dynamic because they include advanced reflection mechanisms and allow to modify the program at run time (e.g., by evaluating the text string as a code, by adding new methods to existing objects or by modifying the type system). Memory management in such languages is almost always automatic, with no explicit memory deallocation.

The current popularity of script languages is connected with their applications in Internet programming. There is a vast number of frameworks facilitating web development for popular script languages, e.g., Catalyst (Perl), Django (Python) or Ruby on Rails (Ruby). They offer a high level of abstraction, introduce convenient design patterns (like MVC) and help with common tasks, such as authentication, form processing or image scaling. Since large web services are created in such technologies, the issue of performance has become increasingly relevant. There is a demand for faster interpreters and many optimisations have been introduced in recent versions of popular script language implementations.

Perl

Perl is a language that was created primarily with text processing in mind. It was originally developed by Larry Wall in 1987. From the very first versions it has supported regular expressions and has contributed their popularization. Perl's regular expression syntax has become an informal standard which is used in many other tools and languages (e.g., Java, JavaScript, Python, Ruby, .NET).

The motto accompanying Perl is: "There is more than one way to do it". This means that the language should not constrain the programmer by forcing him or her to adapt any particular style. By following this principle there is a support for multiple programming paradigms: procedural, functional and object-oriented (although it should be noted that Perl was created with mostly procedural programming in mind and the objects as well as some functional aspects were added later on).

To present Perl intended strengths, I enumerate its features highlighted on the language website [25]:

- object-oriented, procedural and functional,
- easily extendable,
- tools for text manipulation,
- Unicode support,
- database integration,
- C/C++ library interface,
- embeddable,
- open source.

Perl 5 has one canonical implementation written in C. It is a classic interpreter which parses and compiles the source code into an internal representation in byte code and executes that byte code. It is also possible to store a program in precompiled form to reduce its loading time; nevertheless, the interpreter is still needed to execute it.

The lack of alternative implementations can be partially attributed to the lack of formal language specification and partially to difficulties with parsing Perl. Not only does its syntax contain many ambiguous elements, but it also allows some parts of the code to be executed before the rest is compiled, which can influence the whole parsing process. That is why it is often said "Only perl can parse Perl" (where Perl refers to the language and perl to its implementation). Some Perl programs are impossible to parse even with perl. It can be proven that Perl can be described only with an unrestricted grammar, so parsing it requires the use of a Turing machine (not a total Turing machine). As it is impossible to decide whether the Turing machine's calculation will halt, parsing Perl suffers from the halting problem [17].

The long-awaited Perl 6, which is still under development, will have a formal specification and several independent implementations. The most mature of the existing implementations is Rakudo Perl, which compiles Perl 6 to run on a universal Parrot virtual machine.

Python

The Python script language was designed with readability in mind. Its creator Guido van Rossum, started implementing it in 1989 as a successor of the ABC programming language. Python is one of the few programming languages where white space indentation is a part of the syntax. It allows for object-oriented and imperative programming and has some features known from functional languages (e.g., lists comprehension).

According to the Python philosophy: "There should be one—and preferably only one—obvious way to do it". It encourages uniformity and avoids creating alternative ways of doing the same task. Such standards are meant to make the code cleaner and easier to understand for any other Python programmer.

The features listed on the language web page include [9]:

- very clear, readable syntax,
- strong introspection capabilities,
- intuitive object orientation,
- natural expression of procedural code,
- full modularity, supporting hierarchical packages,
- exception-based error handling,
- very high level dynamic data types,
- extensive standard libraries and third party modules for virtually every task,

- extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython),
- embeddable within applications as a scripting interface.

The official Python interpreter is CPython, written in C. It compiles programs into byte code and executes them on a virtual machine. Multiple alternative implementations and modifications of the original one have appeared. Jython and IronPython are Python compilers for the Java Virtual Machine and .NET platform, respectively. Stackless Python is a CPython modification which implements micro threads and does not rely on a normal C stack. Cython compiles a large subset of Python into a standard C code.

It should be noted that most alternative implementations support only Python 2.x, which is no longer being actively developed. Its successor, Python 3, introduced many changes and is not compatible with the previous versions. It is fully supported by official CPython implementation.

Ruby

In 1993 Yukihiro Matsumoto felt that no existing script language actually fulfilled his expectations, so he created a new one—Ruby. He liked some of the features of Perl and Python, but wanted his language to be more object-oriented with object concepts similar to Smalltalk. Ruby introduces some aspects of functional programming through extensive support for anonymous functions known as blocks (that feature, however, was also inspired by Smalltalk).

Ruby is said to follow "principle of least astonishment". This means that the language and standard library behave in such a way as to minimise confusion for experienced users. Following this principle language designers should avoid any kinds of special cases, special syntax, special arguments, etc.

According to the language website [31], the following features are characteristic of Ruby:

- simple in appearance, but very complex inside,
- seeing everything as an object,
- flexibility (possibility to alter parts of the language),
- expressive blocks (closure support),
- extending classes through mixing embeddable modules,
- clean visual appearance.

The main Ruby implementation is referred to as MRI (Matz's Ruby Interpreter). It was originally written as an classic interpreter by Yukihiro himself, but since version 1.9 it has been redesigned and now works as a specialised virtual machine with its own byte code representation and instruction set.

Since 2011 the language has an official specification accepted as an ISO standard. There are several alternative implementations of Ruby, including JRuby (for JVM), Rubinius, IronRuby (for .NET), MacRuby (using Objective-C

run time). All of these offer just-in-time compilation, and MacRuby additionally offers ahead-of-time compilation. Another alternative implementation is Skarb—a Ruby to C compiler developed by the author of this thesis [30].

1.2 Memory management

Manual memory management

In the past the situation where only one program resided in the computer memory at a time was most common. Such programs were usually written in an assembly language and had direct access to the computer memory and hardware. The programmer managed the memory manually and knowing under which addresses the program code was stored and where it was safe to store his data (thos is still so in some cases, e.g., in micro-controller programming).

With the introduction of multitasking operating systems, the standard way of handling memory issues has changed, i.e., it is now the responsibility of the OS to divide the memory between concurrent processes that are generally unaware of one another. When a new process (program) is to be executed, the OS reserves memory space for it. This memory can be divided into three segments: text segment, stack segment and data segment. The text segment simply contains the compiled code of the program. The stack segment is generally used for handling subroutine calls and for storing the parameters and local variables associated with them. The data segment contains global variables and the so-called heap. The heap is general storage for any data which were not declared statically in the program code. It has some initial capacity, but it can be extended later on during program execution through system calls (**sbrk** in Unix systems, **VirtualAlloc** in Windows). Although it can be managed manually, just as in the case without OS, it is usually accessed through a memory allocator.

The memory allocator is a set of functions responsible for reserving and freeing the heap memory in an orderly fashion and for hiding unnecessary details from the programmer (in C those are the **malloc**, **realloc** and **free** functions). The allocator keeps track of the addresses which were already reserved and assures that they can be effectively reused after they are freed. While doing this, it tries to reduce fragmentation, which renders the memory unusable, as well as to preserve the locality of successively allocated memory chunks to improve execution performance. The default allocator is usually included in libraries provided by the operating system, nevertheless, it is possible to write a custom one. Performance of the memory allocator plays a significant role in the overall performance of the software. Choosing an allocator is often a choice between more economic memory usage and faster program execution. It should be noted that although using an allocator makes the work of the programmer considerably easier, as long as there is explicit memory deallocation (**free** call) involved it is still called manual memory management.

Automatic memory management

In the case of bigger programs memory management using functions such as **malloc** and **free** resembles complicated bookkeeping. It is not unusual for the programmer to forget to deallocate the memory, thus creating a memory leak, to attempt to deallocate it twice or use to a pointer which no longer points to anything. To address this problem, automatic memory management was introduced. Basically, it requires that the programmer only allocate memory. Deallocation takes place automatically when the data residing in the allocated memory chunk are no longer needed—this process is called garbage collection.

The issue here is how to determine which data are not needed. A widely accepted method is checking memory reachability. If at a certain point during program execution the chunk of memory can be accessed through a pointer traversal starting from any reachable local or global variable, the data residing in that memory can be potentially needed. However, this does not mean that it will be used. This is the reason why in some cases a programmer relying on the garbage collection mechanism has to explicitly nullify a reference to be sure that certain pieces of memory can be reclaimed before the program exits.

Reference counting

The simplest garbage collection algorithm is based on reference counting. In this approach each allocated chunk of memory is associated with a dedicated counter which counts the number of references (pointers) referring to that memory chunk. Each time a reference is destroyed or overwritten, the applicable counter is decremented; similarly, copying or creating a reference leads to a counter increment. When the reference count of a memory chunk becomes zero, the chunk can be safely deallocated.

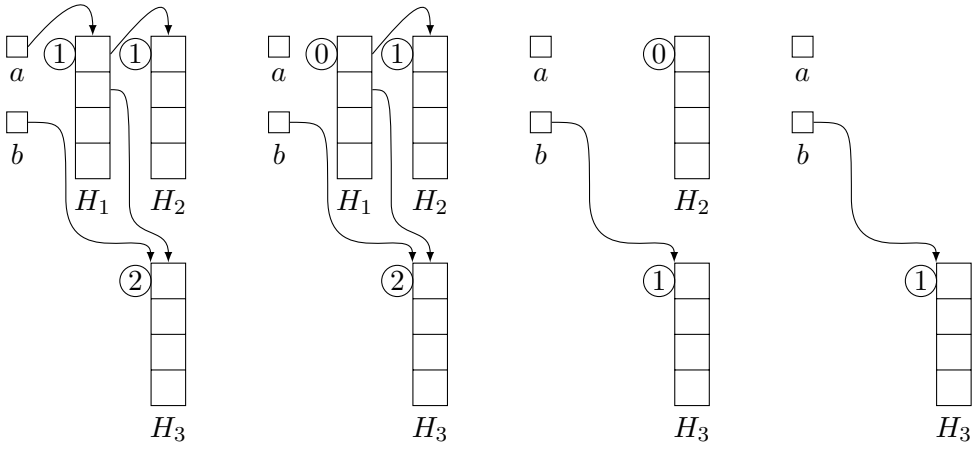

Figure 1.1: A simple scenario of garbage collection using reference counting. a and b represent local pointers, while H_1, H_2 and H_3 are compound objects allocated as continuous memory chunks on the heap.

A method of garbage collection using reference counting is illustrated in Figure 1.1. During the first step variable a is overwritten, which causes a decrement of the reference counter of object H_1. As its value becomes zero, it is possible to destroy the whole object. There are references to H_2 and H_3 stored in H_1, so it is necessary to update their counters as well. It may happen that H_2 can also be destroyed but H_3 is still needed. This scenario clearly shows that destroying a single reference can lead to the destruction of multiple objects.

Apart from simplicity, the huge advantage of reference counting garbage collection is that the memory is reclaimed as soon as it is not needed (I will later show that it is different with other garbage collection mechanisms). The programmer is fully aware when this happens and has a certain degree of control over it. Reference counting can be used with most standard memory allocators.

On the other hand, there are some important drawbacks of this memory management technique. First of all, it is necessary to reserve additional space for the reference counter along each memory chunk which increases memory consumption. Then, in a naive implementation, each operation on the references leads to counter updates, even if this does not result in memory deallocation—these operations add an additional cost to the program execution. Finally, it is impossible to free cyclic structures by using reference counting because counters associated with the memory chunks of such a structure will never reach zero. To deal with cycles an additional mechanism is needed, which will increase algorithm complexity.

The tracing garbage collector

A slightly more sophisticated class of garbage collection algorithms are the tracing garbage collectors. Such a garbage collector does not check object reachability in real time, i.e., after every operation that could change it, but instead it works in cycles. The cycle typically starts when the garbage collector is notified that there is a need for more memory. During the cycle the memory is traversed to determine which objects are still reachable and which can be safely deallocated. The traversal is guaranteed to visit all reachable objects but does not follow reference cycles.

The tracing garbage collector has one huge advantage in comparison with reference counting: it can handle reference cycles without any additional mechanisms. Also, there is no overhead associated with pointer operations. However, it lacks the predictability that is characteristic of reference counting: the programmer generally does not know when the garbage collection cycle will occur and when unused objects will be deallocated. Naive implementations of tracing garbage collectors work in stop-the-world manner. This means that the execution of the whole program is halted until the collection cycle is finished. This is generally undesirable behaviour, especially in interactive or real-time applications.

There are many kinds of tracing garbage collectors, each with its own characteristics. Richard Jones, in his book on garbage collection [15], describes two approaches as classical ones: mark-and-sweep and the copying garbage collector. I shall describe them shortly.

Mark-and-sweep collector

With the mark-and-sweep algorithm each collection cycle consists of two phases. The first one is the *mark* phase, during which the memory is traversed following a program data structures. The traversal starts from the so-called roots, i.e., pointers immediately accessible at the current point of program execution. The set of roots typically includes all local variables (residing on the stack), global variables and values stored in CPU registers. The algorithm visits all objects accessible from the roots, then objects accessible from newly visited objects, etc. All visited objects are marked, which usually means setting an appropriate bit in the object's structure or some additional structure used by the garbage collector. An already marked object cannot be marked for the second time and any objects referenced by its fields are not visited repeatedly—this guarantees that the traversal will eventually end. The second phase of the algorithm is the *sweep* phase. It includes some form of heap memory traversal and reclaiming of unmarked objects.

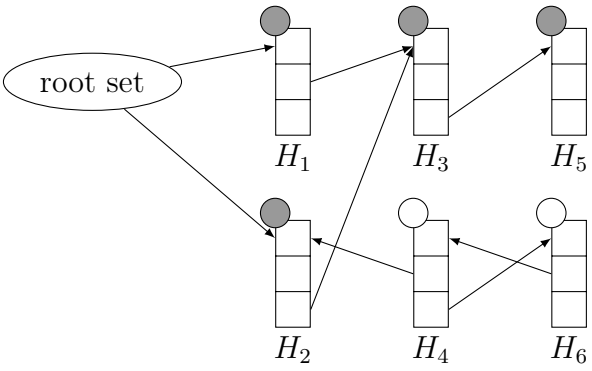

Figure 1.2: Heap objects after the mark phase. H_i are objects allocated on the heap. The circle in the top left corner of an object represents the mark bit.

A simple scenario of the mark-and-sweep garbage collector is presented in Figure 1.2. It is visible that after the mark phase, H_4 and H_6 are left unmarked and can be deallocated. As opposed to garbage collection based on reference counting, with the mark-and-sweep method H_4 and H_6 can be deallocated in any order since the reference from H_4 does not affect H_6 reachability.

The mark-and-sweep garbage collector can generally be used with any memory allocator, however, in most cases it is used with a customised allocator that optimised to work specifically with certain garbage collector implementation. Such a garbage collector can be universal and can work without any special

information stored at compile time when it is implemented as a *conservative* collector. This means that the garbage collector scans all of the reachable memory (CPU registers, program stack, thread stacks) and treats everything that can potentially be a pointer as a pointer. This can lead to a situation when an ordinary numeric value is treated as a valid pointer and prevents some objects from being freed. However, it was experimentally demonstrated that this is quite unlikely and does not introduce successively growing memory leaks.

Copying collector

The second presented approach is the copying garbage collector. It divides the memory into two subspaces: reachable objects are kept in one of them and the other is considered to be empty (i.e., containing only garbage). All new objects are allocated into the first one. When the collection cycle starts, a pointer traversal similar to that from the mark phase of the mark-and-sweep collector takes place. The difference is that the objects, instead of being marked, are copied to the other memory subspace. After such a traversal the roles of the subspaces are swapped—the first one is now considered empty and the second contains only reachable objects (and free space for new objects).

The copying garbage collector needs a special memory allocator and demands that any object be moved at run time. It generally cannot be implemented in a conservative manner. This makes such garbage collectors less universal than the mark-and-sweep ones.

Memory management in Perl

There are three basic data types in Perl: scalars, arrays and hashes. All of them are heap-allocated. The memory is managed with reference counting with a heavy accent on memory reuse. This means that in many cases a variable whose reference counter has reached zero is not not truly freed but instead is cleared and kept for later usage.

Perl supports weak references which have the same features as normal references but do not increment the object reference counter. In other words, a weak reference is not enough to keep an object alive. There is also an **undef** function which allows to explicitly undefine a variable; however, it does not free the memory, it only cleans it.

There is no mechanism to free reference cycles in Perl. Structures in such cycles are deleted only after the process exits. Naturally, it is possible to break a cycle explicitly (e.g., by using weak references or undefining variables).

This memory management schema is well-suited for the needs of short scripts performing batch processing or for interactive programs that are not processing any bigger data. However, in long-living processes the accumulation of garbage trapped in reference cycles may lead to a huge memory bloat.

Memory management in Python

In CPython, i.e., the official Python implementation, virtually all objects are allocated on the heap and garbage is collected through reference counting. Only pointers to heap-allocated objects are stored on the stack. Numbers and character strings are also represented as objects. The language supports object finalisers which are called when the object is to be deleted because its reference counter reached zero. With reference counting it is common that multiple objects are deallocated at once. Python guarantees that they are reclaimed in topological order, i.e., when an object's finaliser is called the objects referenced in its fields still exist. Python supports weak references just like Perl.

Reference counting does not handle cases with cycles in references. To address this problem an additional mechanism based on a tracing garbage collector was introduced. It employs a special tracing garbage collector algorithm to analyse all objects that were not deleted by the reference counting mechanism and can detect cyclic references. A cycle that is unreachable from the outside can be safely deleted, assuming that the objects from the cycle do not have finalisers. It is impossible to delete cycles with finalisers because the order in which the finalizers should be invoked cannot be determined. The garbage collector puts such objects on a special list and expects that the programmer will deal with them by breaking the cycle manually.

The tracing garbage collector performs the collection periodically after a specified threshold of allocations and deallocations is exceeded (it is not connected with current memory usage). It works in a stop-the-world manner, so the whole program execution freezes until the collection is finished. This can be a problem in interactive applications, thus there are various attempts to deal with it. The garbage collector interface allows the programmer to change the threshold of the allocations and deallocations or to force the collection to be performed at some point. However, sometimes the best solution is to avoid reference cycles in the program altogether (e.g., by using weak references) and to turn off the tracing garbage collector.

Alternative Python implementations, such as Jython and IronPython, rely on the garbage collectors of their run time environments. PyPy uses a custom tracing garbage collector for all objects. The developers of these alternative implementations have problems with replicating CPython behaviour connected with garbage collection—the most important matter are finalizers, which, in CPython, are called as soon as an object is not needed and, in most other implementations, in some undefined future.

Memory management in Ruby

Just as in Python, in Ruby there are no non-object values. Consequently, in the MRI all objects are allocated on the heap and only the pointers are stored on the stack. There is one notable exception to this rule—small integers (represented by Fixnum instances) are represented directly by their pointers. In

every pointer a special bit indicates whether it is really pointing to anything or whether it should be treated as an integer. This is only a matter of optimisation and does not affect Fixnums behaviour in any way.

The MRI uses tracing garbage collector to the manage memory. The collection process is executed when there is no memory to allocate a new object to. Because of the way the MRI communicates with C extensions (functions written in C extending the functionality of Ruby), the garbage collector has to work in a conservative manner. It also acts in a stop-the-world fashion, which can introduce pauses during program execution.

In Skarb, Ruby to C compiler, memory management strategy is very similar. It uses the Boehm-Demers-Weiser garbage collector—an open source conservative garbage collector for the C language. [3] A notable difference is that BDW GC works iteratively: instead of stopping execution of the program completely until the full collection is done, it attempts to split the work into parts and does a bit with every allocation. It should make the long pauses less probable.

1.3 Cost of memory management

An important question to be asked is the question of memory management costs. These costs include two factors: the memory consumption overhead (i.e., the portion of memory reserved for the process but not used to store any useful data) and CPU time spent in memory management routines. Due to the recent technological advances computer memory has become relatively cheap. If a program consumes too much memory in most situations installing an additional memory module is an affordable solution. As other computer compontents are more expensive, reducing computation time throught purchasing new hardware is often impossible. Because of this, execution time overhead seems currently more important and I will focus on it.

The question is how much time does an executed process spend on managing its own memory instead of doing work that is useful from the user's perspective. The costs of operations needed for memory allocation, deallocation and, in case of automatic memory management, identifying unused memory chunks should be included. However, the costs of functions reserving additional memory for the process and extending its heap are excluded. This is work done by the operating system and is not a subject of optimisation in a language interpreter.

With memory management based on malloc and free, all that has to be done is to measure the time spent in these functions. With garbage collection based on reference counting, the time spent on counter updates is to be included. In the case of a tracing garbage collector, the cost of the collection cycle should be included instead.

Experiment with popular C/C++ programs

There is a classical study by Detlefs et al. [7] which compare the allocation costs of various memory allocators and the contemporary version of the Boehm-Demers-Weiser garbage collector. The authors used profiling tools to measure the time spent on memory allocation in popular C and C++ programs. Selected programs included:

- Xfig—an interactive vector graphic program,
- Make—a utility for automation of building executables from source files,
- Gakw—a GNU AWK interpreter,
- GhostScript—a Postscript interpreter.

The time spent on the internal memory management mechanisms of AWK and Postscript was not included. With Xfig the test case consisted of drawing simple geometrical shapes, replicating and deleting them. Other programs were processing input files typical for them: a large makefile, a text formatting script and 126-page user manual, respectively.

	BDW 2.6	libg++ 2.4.5
Xfig	34.5%	3.5%
Make	14.3%	3.5%
Gawk	65.5%	11.9%
Ghost	54.8%	6.5%

Table 1.1: Percentage of execution time spent on memory management in C and C++ programs.

Table 1.1 is an excerpt from the results of the cited study. BDW is compared with the malloc implementation from the GNU C++ library, which later evolved into Doug Lea malloc. [20] Unsurprisingly, with a tracing garbage collector the percentage of time spent on memory management routines was much higher than with a standard malloc. The time spent on managing memory always constituted a substantial portion of the total execution time, however, the exact numbers varied greatly depending on which program was analysed. These results do not necessarily show the advantage of malloc/free memory management over tracing garbage collector, but they do signalise that the performance of the garbage collector has a huge impact on the performance of the program.

Research on Lisp, ML and Smalltalk

Many articles were written on garbage collection in functional languages such as Lisp or ML. These observations are somewhat relevant because both Python

and Ruby have incorporated functional language features, such as continuations, which lead to more intensive heap usage. Diwan et al. [8] benchmarked six SML/NJ compilers and concluded that garbage collection accounts for 19% to 46% of total program run time. Classic studies by Steele [29] and Ungar [32] reported that the average time spent on memory management in contemporary Lisp interpreters was around 30%. In the same work Ungar [32] presented a garbage collection algorithm for Berkeley Smalltalk which accounted for only 2% of application run time. Chambers [5] pointed out that Ungar's experiments were biased because he linked a highly optimised garbage collector implemented in an assembly language with a relatively slow and bulky Smalltalk interpreter. The fraction of time used for memory management was extremely low because the other operations were done less efficiently. Chambers also predicted that with an advance in optimising compilers and interpreters, the fraction of time spent on memory management in program execution will rise.

Experiment with modern script languages

I am unaware of any recent study analysing the costs of memory management in the interpreters of Perl, Python or Ruby. Therefore, I conducted a small experiment myself based on simple programs from an older version of the Computer Language Benchmarks Game [10]. Using the gperftools [11] profiler and its Ruby version, perftools.rb [12], I measured the portion of time spent on memory management in both the Ruby and Python programs. The run time environments being tested included Python 3.2.3, Ruby 1.9.3 and Skarb. The first two are official languages implementations and the third is an experimental Ruby to C compiler developed as a part of my bachelor thesis. Notable difference between Skarb and the other environments is that it aims for compilation to C and then to native machine code instead of real-time interpretation.

As was mentioned before, Skarb uses a BDW garbage collector (currently in version 7.1) and Ruby relies on its own garbage collector implementation. Python's hybrid memory management strategy (reference counting and tracing garbage collector) made measurements a little harder. For the profiler to take into account all memory management operations, Python's source code had to be slightly modified. In the original version the reference counters are updated through simple macros—I transformed them into inline functions. From the performance perspective they should be equivalent, but the inline function calls can be tracked by the profiler.

Test cases included:

- Ackermann—calculating the value of the Ackermann function,
- Matrix multiplication—the multiplication of large matrices containing floating-point numbers,
- Quicksort—sorting large arrays of floating-point numbers with Quicksort,
- Binary trees—allocating and deallocating large number of binary trees.

Calculation of the Ackermann function was done both in integer numbers and floating-point numbers to ensure that values would be allocated as objects in Ruby 1.9. Programs were implemented in both languages using the same strategies and were to be as similar as the language differences allowed.

Each test program was executed five times and the mean value was calculated from the collected values. C source code generated by Skarb was compiled with gcc 4.7 compiler. Programs were executed on a computer with AMD Phenom II X2 555 3.2 GHz processor, the operating system was 64-bit version of Arch Linux.

	Skarb	Ruby 1.9.3	Python 3.2.3
Ackermann (integer)	55.3%	0.0%	25.5%
Ackermann (float)	71.2%	37.8%	22.7%
Matrix multiplication	31.5%	47.4%	22.9%
Quicksort	55.4%	25.1%	27.1%
Binary trees	56.4%	27.5%	48.2%

Table 1.2: Percentage of execution time spent on memory management in both the Ruby and Python programs.

The results are presented in Table 1.2. As one can see, all programs operating on large sets of data use the heap intensively and a large fraction (over 20%) of their execution time is spent on memory management.

An interesting case is the Ackermann function. Its natural implementation in a language such as C would not use the heap at all. The same applies to its integer version in Ruby. Since the original Ruby implementation treats small integer numbers as immediate values, no allocation or deallocation takes place. However, when the calculation is done in floating-point numbers, each value is represented as a separate heap object which needs to be allocated. During the calculation of the Ackermann function many small objects are created—the heap is used intensively. In both Skarb and Python both floating-point and integer values are represented as objects, so both versions of the Ackermann function manage memory in a similar fashion.

Unfortunately, because of the complexity of Perl 5 internals, it was impossible to do a similar test for Perl without bigger modifications of the source code. However, similarities in the object system and memory management strategies between Perl and Python provide a basis for the assumption that Perl programs would also spend considerable time on the memory management.

As we can notice, with modern systems and runtime environments memory management still consumes large fractions of program execution time. In languages, which treat all values uniformly as heap-allocated objects even simple computation can become memory intensive tasks. Any optimisations of the current memory management techniques would improve the overall performance of script languages.

Chapter 2

Memory management optimisations

Various techniques were proposed to improve the performance of automatic memory management mechanisms. The most straightforward way is to improve the garbage collection algorithm. Much research was done in this area, which lead to the development of heuristics such as generational garbage collection, where objects are segregated into generations depending on their lifespan [21, 32, 28, 27]. Many optimised modifications of basic algorithms described in the previous chapter were proposed. The choice of algorithm is highly dependent on the language and run time environment.

Not only the garbage collector, but also the memory allocator can be a subject of optimisation. As expected, multiple allocator algorithms with distinct characteristics exists [33]. Choosing the best one under the given circumstances is not a trivial task. The approach suggested in practical books for programmers [4] was to construct custom memory allocators for specific classes of objects. This would allow to make some optimisations based on additional information on the object structure. On the other hand, using custom allocators may lead to additional memory fragmentation and may make global memory optimisations impossible. Experiments conducted by Emery Berger et al. showed that in many cases custom allocators perform worse than optimised general-purpose allocator [1].

Another kind of possible optimisations are optimisations based on data locality. They stem from the notion that the program's performance is limited by both memory access time and page translation costs. To reduce these costs and to hide access latency, mechanisms such as the translation lookaside buffer and multi-level caches were introduced. However, it was suggested that they are not fully utilised because of poor program data layout. An effort was made to improve this situation with specific modifications of the garbage collector [16].

Another class consists of various optimizations based on static code analysis. They are generally complementary with the standard techniques and, to a great extent, independent of the garbage collector mechanism being used. I will include the following optimisations in this class:

- Stack allocation—identifying objects local to a function and allocating them on the stack.
- Region allocation—identifying groups of objects with similar lifetime and allocating them in a single memory region which can be free with one instruction.
- Scalar replacement—replacing objects with simple values or storing object data in CPU registers.
- Object reuse—reusing the dead objects' memory for newly created objects, often referred to as compile time garbage collection.

Among all these techniques, stack allocation and object reuse are the most universal. They do not require any in-depth knowledge of the run time environment or the garbage collector. As opossed to the optimisations based on data locality or scalar replacement, they are machine and system independent. If a conservative garbage collector is used, it is possible to implement these optimisations without modifying existing memory management mechanisms. Because of those merits, I have chosen stack allocation and object reuse as potentially promising techniques and decided to focus on them.

2.1 Stack allocation

In languages such as C all values and structures created inside a function are typically allocated on the stack. The only way to allocate memory on the heap is to use explicit allocation instructions. In C++ structures becomes objects, which are also allocated on the stack by default and can be allocated on the heap using the *new* operator. In languages such as Java, all objects are created using the *new* operator and in most environments are heap-allocated by default. The interpreters of languages such as Ruby or Python allocate even simple numerical values on the heap, and the stack contains only object references. The idea behind stack allocation optimisation is to change this behaviour and to allocate some objects on the stack.

The sole process of stack allocation is very simple. Typically, a stack frame consists of a few blocks, as is presented in Figure 2.1. The topmost block is the region of memory reserved for local variables. To reserve more memory dynamically at run time, all that has to be done is to move the stack pointer. Naturally, only the current (topmost) stack frame can be extended this way. This way of reserving memory is very fast and does not introduce memory fragmentation. All reserved memory is reclaimed when the function exits.

Stack-allocated objects are allocated on the stack frame of the function in which they are created. As they are reclaimed automatically when the function exits, only objects that do not escape the function context can be allocated on the stack. To identify them automatically, i.e., without additional declarations from the programmer, some form of escape analysis is needed. Various works

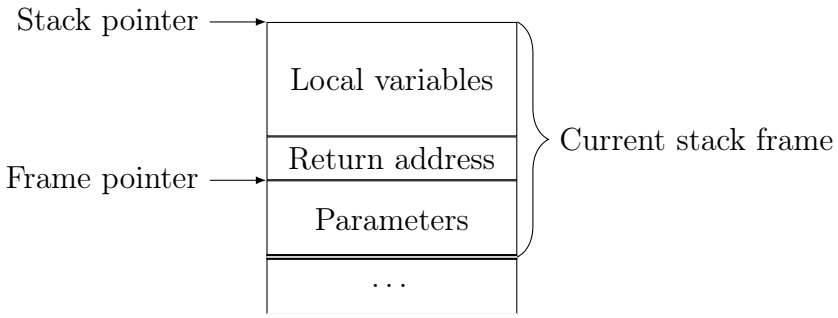

Figure 2.1: Typical layout of a stack frame.

investigating the possibility of stack allocation have focused on finding the best algorithms for escape analysis.

The term *escape analysis* was probably used for the first time in an article by Young Gil Park and Benjamin Goldberg [24]. They analysed the escape state of list cells created in functional languages. Based on escape analysis, they proposed stack allocation and region allocation optimisations as well as in-place object reuse (a form of compile time garbage collection described in the next section). Their work remained theoretical and no prototype system was implemented.

Thomas Kotzmann and Hanspeter Mössenböcket explored the possibilities of stack allocation and scalar replacement in the context of dynamic compilation of the Java code [19]. They presented a fast escape analysis algorithm and implemented it for the Java HotSpot compiler. Because they were examining dynamic compilation, they focused on analysis speed rather than precision.

Jong-Deok Choi et al. proposed a connection graph abstraction, which captures the relationship between objects and pointers, as a means for precise escape analysis [6]. They have implemented this method in a static Java compiler and have used the obtained information for stack allocation and elimination of synchronisation operations in a multi-threaded context.

Bruno Blanchet implemented escape analysis and stack allocation optimisation for the Java-to-C compiler turboJ [2]. In his variant of escape analysis he used a form of graph abstraction, but used integers to represent graph paths. Such an implementation made escape analysis remarkably faster. He presented benchmarks that showed that stack allocation can decrease program execution time by up to 43%.

2.2 Compile time garbage collection

Commonly used garbage collection mechanisms tracing object reachability at run time were described in the previous chapter. I will refer to them as to run time garbage collection (RTGC). The complementary mechanism which finds dead objects using static analysis is called compile time garbage collection

(CTGC). Memory allocated for objects which are certainly dead at some point of program execution can be reused for newly created objects. Such a mechanism reduces the number of allocations and total memory usage of the program. It should also speed up program execution because it makes the run time garbage collector cycles occur less frequently.

To perform compile time garbage collection, some kind of static program analysis is needed in order to identify dead objects. The analysis can be relatively simple and based on a few identified expressions leading to the creation of a short-lived object or remarkably complex, based on a comprehensive program abstraction. However, even the most complex form of static analysis is never fully precise because it lacks information available only at run time. That is the reason why CTGC can only be a complementary method and cannot substitute RTGC.

Escape analysis performed for the purpose of stack allocation is in fact a special case of object liveness analysis performed for CTGC. In escape analysis, the goal is to identify objects that outlive the function in which they are created. In CTGC analysis the goal is to identify objects that die before other objects are created.

Although the concept of CTGC is quite old, it is still rarely encountered in modern compilers. In the past the possibility of compile time garbage collection was analysed mostly in the context of functional and declarative languages such as Lisp and Prolog [13, 18, 23]. There is also an article on CTGC in Smalltalk [26] which describes in fact the realisation of a stack allocation concept.

In more recent years a practical CTGC system was developed for the Mercury language [14, 22]. Mercury is a strongly typed logic programming language based on Prolog. Its syntax allows to perform very precise static analyses of object liveness. The developed CTGC system identifies dead objects and allows to reuse them either in the same function or across functions. The presented benchmarks showed major (up to 50%) memory consumption reduction and noticeable speedups (up to 15%).

Chapter 3

Optimisations implemented in the Skarb compiler

I have decided to adapt the approach described in [6] and to implement a mechanism for automatic stack allocation for the Skarb compiler [30]. Skarb was chosen because of the relative simplicity of its internals and the fact that I already had in-depth knowledge concerning its design. This made any necessary modifications considerably easier to make than would be in the case of a different compiler. Furthermore, a clear distinction between the compilation and execution phase, which is present in a classic static compiler such as Skarb, allows to get the most profit out of such optimisations. Although Skarb currently accepts only a subset of the Ruby language, it is mature enough to test various optimisations and to check if they introduce any significant performance gain.

In this chapter I describe a detailed procedure of building a connection graph abstraction for Ruby programs. I present how this kind of abstraction can be used for escape analysis. Then details on the implemented stack allocation optimisations relying on escape analysis follow. Finally, I introduce another kind of optimisation based on a connection graph—a mechanism for reusing memory allocated for local objects. All described optimisations are newly introduced mechanisms previously absent in Skarb.

3.1 Connection graph abstraction

The connection graph is a form of program abstraction introduced by Choi et al. [6] as a means for escape analysis. The connection graph allows to establish reachability relationships between objects and object references. After such relationships are established, it is easy to identify which objects can only be reached through local references. I will call these objects *local objects*. They cannot be reached from outside the method so they do not *escape* the method.

Normally, when speaking about objects what is meant are the run-time objects, i.e., objects created during program execution. In a connection graph

the objects which are represented are compile-time objects. Generally, each compile-time object is connected with a single object constructor call in the program code. In this thesis the run-time objects will be called *concrete objects* and compile-time objects will be called *abstract objects*. A single abstract object can represent multiple concrete objects because its corresponding constructor call statement can be executed multiple times (e.g., in a loop).

Definition 1. *A connection graph is a directed graph* $CG = (V_o \cup V_r, E_f \cup E_r \cup E_d \cup E_p)$*, where:*

- $V_o = V_{os} \cup V_{oph}$ *is a set of object nodes.*
  - V_{os} *is a set of standard object nodes representing known objects (i.e., those which can be connected with a specific constructor call statement).*
  - $V_{oph} = V_{op} \cup V_{of} \cup V_{og}$ *is a set of phantom object nodes, representing unknown objects which may or may not exist*
    - V_{op} *is a set of nodes representing the formal parameters of a method.*
    - V_{of} *is a set of nodes representing the unknown values of an instance variable (object field).*
    - V_{og} *is a set of nodes representing the unknown values of a global variable (static field).*
- $V_r = V_{rl} \cup V_{rf} \cup V_{rg} \cup V_{re}$ *is a set of reference nodes.*
  - V_{rl} *is a set of nodes representing local variables.*
  - V_{rf} *is a set of nodes representing instance variables (object fields).*
  - V_{rg} *is a set of nodes representing global variables or class variables (static fields).*
  - V_{re} *is a set of nodes representing references returned by expressions such as function calls, conditional statements, etc.*
- E_f *is a set of field edges. A field edge denotes that an object has a certain field.* $p \to q \in E_f \Leftrightarrow p \in V_o \wedge q \in V_{rf}$
- E_r *is a set of reference edges. A reference edge denotes that an object may be reachable through certain reference.* $p \to q \in E_r \Leftrightarrow p \in V_r \wedge q \in V_o$
- E_d *is a set of deferred edges. A deferred edge denotes that a reference points to the same object as another reference.* $p \to q \in E_d \Leftrightarrow p \in V_r \wedge q \in V_r$
- E_p *is a set of placeholder edges. A placeholder edge denotes that a phantom object node takes place of an unknown initial value of an instance variable.* $p \to q \in E_p \Leftrightarrow p \in V_{of} \wedge q \in V_{rf}$

The main difference between standard object nodes and phantom object nodes is that in the case of a phantom object it is impossible to connect it with any specific constructor call. The existence of such an object is only deduced from the context. Phantom objects can be merged with real objects at some point of execution. The details are given in the subsection concerning method calls.

Deferred edges do not introduce any information relevant from the perspective of escape analysis. They exist only to reduce the number of edges created during the graph updates after the assignment statements. The details are given in the subsection concerning assignments.

Each translated method has its own connection graph containing objects reachable from this method. The methods are identified by the owner, name and types of the arguments passed. The nodes of the connection graphs are identified by their unique names. Name of a V_{rl} or V_{rg} node is simply the name of the variable. The name of a V_o or V_{re} node is an automatically generated numerical id with a prefix depending on the node type. The name of the V_{rf} node is the name of the object node combined with the name of the field. Two nodes from different connection graphs with the same name are treated as the same node occurring in different contexts. Such correspondence between nodes of separate graphs is used in the interprocedural analysis, which is described in later sections.

Definition 2. $\textsc{Fid}(v)$ *is the name of an instance variable (object field) represented by field node* $v \in V_{rf}$.

Definition 3. $\textsc{Owner}(v)$ *is a node from* V_o *representing the owner of the field represented by node* $v \in V_{rf}$. $p = \textsc{Owner}(v) \Leftrightarrow p \rightarrow v \in E_f$.

Escape analysis

Based on a method's connection graph, it is possible to perform escape analysis by following three simple rules:

1. Each node has an escape state: *no escape*, *arg escape* or *global escape*. The initial state is *no escape*.
2. The state of nodes representing an object passed to a function as an argument or a value returned from the function should be set to *arg escape*. The state of nodes representing global variables should be set to *global escape*.
3. If a node's escape state is set to *arg escape* or *global escape*, then the nodes pointed to by its outgoing edges should also have their escape states set to the same value.

After the escape state is propagated in this manner, every object in *no escape* state is recognised as a local object.

```
def foo(p)
   a = 1
   b = "abc"
   a = p
   return b
end
```

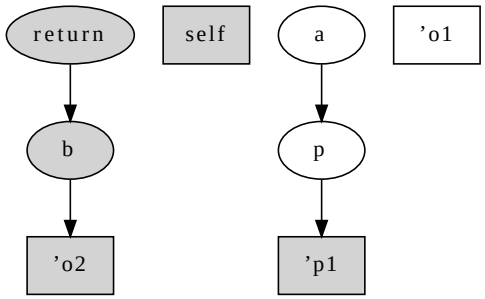

Figure 3.1: Connection graph for the method *foo*.

Definition 4. ESCAPE(v) *is an escape state of node* v.

Figure 3.1 presents a simple connection graph for method *foo*. The rectangles represent object nodes and the ellipses represent reference nodes. The node colour represents the escape state: white is for *no escape* and grey for *arg escape* or *global escape*. As one can see, there is an object node *self* representing an implicit argument—the object being the method owner.

The escape analysis performed with the help of a connection graph does not need to be of an absolute precision. It can be allowed for some local objects to be mistakenly treated as objects which escape the method. What is important is to make sure that the opposite will not happen: no escaping object should ever be treated as a local one.

Intraprocedural analysis

The connection graph is built incrementally and updated after each translated statement. In the next sections I describe the way each kind of language statement is modelled in a connection graph. Since any statement can be treated as an expression returning value, information about a graph node representing that value is attached to the statement. This information is used while modelling compound expressions, which take sub-expressions as arguments.

The proposed way of building a connection graph is both flow-sensitive and path-sensitive. This means that the order of program statements and conditional branching is taken into account. In fact, in each conditional branch a different connection graph is built, and when the branches converge these graphs are merged. The details are given in the paragraphs concerning conditional blocks.

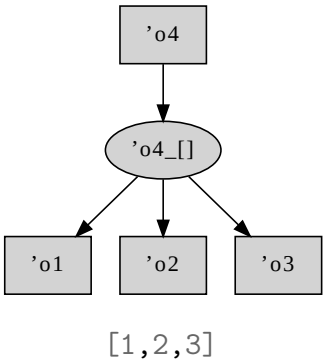

Figure 3.2: Modelling an implicit array constructor.

Object creation

```
Object.new, 1, 1.1, "abc", [1, 2], {"a" => 1}
```

In Ruby there are three types of statements leading to an immediate object creation: call of a *new* method of any class object[1], the use of numeric or string literal or the use of a special form of an array or hash constructor. Each of these statements leads to the creation of a new object node in a connection graph. This node represents the return value of the statement. In the case of an implicit array or hash constructor, a special indexer field node is created for the object and it is linked with the nodes representing values of arguments. It is presented in Figure 3.2.

Assignments to variables

```
a = 1, @a = 1, @@a = 1
```

The modelling an assignment consists of a couple of steps. First, it is necessary to check if the reference node representing a variable is present and to create it if this is not. The second step is the so-called bypass operation which replaces deferred edges with appropriate reference edges. The pseudo-code for this operation is presented in the algorithm block 1.

After bypass of the node v is performed, all outgoing edges of v should be deleted. Then the actual assignment can be modelled by linking the variable node with the node representing the value of the expression on the right side of the assignment. If that value is represented by an object node new than a reference edge is created. If it is represented by an another reference node than a deferred edge is created instead.

An example of a connection graph modification during an assignment is presented in Figure 3.3. After the first two assignments, reference node a points

[1]Normally this method could be overridden, but the current version of Skarb does disallow this. It can be safely assumed that *new* method will always create a new object.

Algorithm 1 The procedure replacing deferred edges incoming to a node with appropriate reference edges.

```
function BYPASS(v)
    for all p ∈ V_r : p → v ∈ E_d do
        for all q ∈ V_r : v → q ∈ E_d ∪ E_r do
            Add edge p → q.
        end for
        Delete edge p → v.
    end for
end function
```

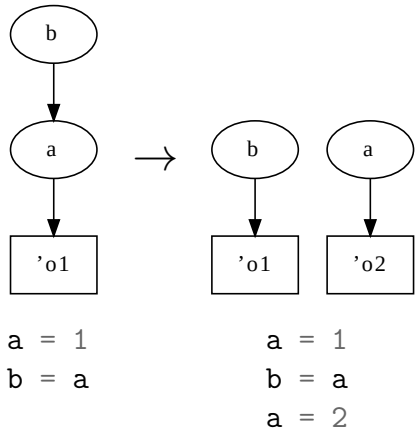

Figure 3.3: Modelling assignment to a local variable.

to an object node and reference node b points to a. Then, after the third assignment a BYPASS operation is performed for node a, and node b is linked directly with the object node $'o1$. Node a points to a the newly created object $'o2$.

Three kinds of assignments can be distinguished: assignment to a local variable, assignment to an instance variable and assignment to a global variable. Assignments to local variables are modelled exactly as described in the previous paragraph. Assigning to an instance variable requires additionally assuring the existence of an edge between the *self* object node and the field node (it is always the *self* node because in Ruby it is impossible to affect an instance variable from the outside of the object, i.e., all instance variables are private). When assigning to a global variable, the node representing the value of the right-hand expression has its escape state set immediately to *global escape*. This is a simplification which allows to avoid the tracking of values of global variables during the execution of all functions.

Referring to a variable

`a`, `@a`, `@@a`

Referring to a variable in a program code can be modelled as a simple statement whose return value is a reference node corresponding to this variable (if such a node does not exist, it should be created). Furthermore, it should be checked if this reference node has any outgoing edges. If it does not then there are two possibilities: the variable points to the *nil* object or to an unknown object created outside of the method. In the case of a local variable only the former can happen. In the case of an instance variable or a global variable, the latter safely can be assumed. To model a such situation a new object node has to be created and linked with the variable node. Its escape state should be set to *arg escape* or to *global escape*.

With a global variable the created object node is the phantom object node $g \in V_{og}$. With an instance variable it is node $n \in V_{of}$. Every node from V_{of} is connected with a placeholder edge to an instance variable node. If $v \in V_r$ is a variable node, then edge $n \to v \in E_p$ is added. To make the relationships between nodes easier to follow, I will introduce a definition:

Definition 5. INITIATOR(n) *is an instance variable node* $v \in V_{rf}$ *which initiated the creation of* $n \in V_{of}$. INITIATOR$(n) = v \Leftrightarrow n \to v \in E_p$

Conditional blocks

`while`, `if`, `case`

A conditional block is a group of instructions inside a loop or a single branch of a conditional statement (*if* or *case*). Opening a new block results in creating a copy of a current connection graph. Statements executed inside the block affect only the copy of the original connection graph. When the block is closed, the two graphs $CG' = (V', E')$ and $CG'' = (V'', E'')$ are merged into a new graph $CG = (V' \cup V'', E' \cup E'')$. Then, a special update operation is performed on the nodes of the new graph to change the deferred edges into reference edges if the node at the end of a deferred edge points to something else in the new graph. The pseudocode for this operation is presented in algorithm block 2.

Algorithm 2 Procedure of updating a connection graph after merging a conditional block.

```
for all v ∈ V_r do
    if {p ∈ V : v → p ∈ E'} ≠ {p ∈ V : v → p ∈ E''} then
        BYPASS(v)
    end if
end for
```

After the update, graph CG becomes the main connection graph representing the pointers and object relationships after execution of the block.

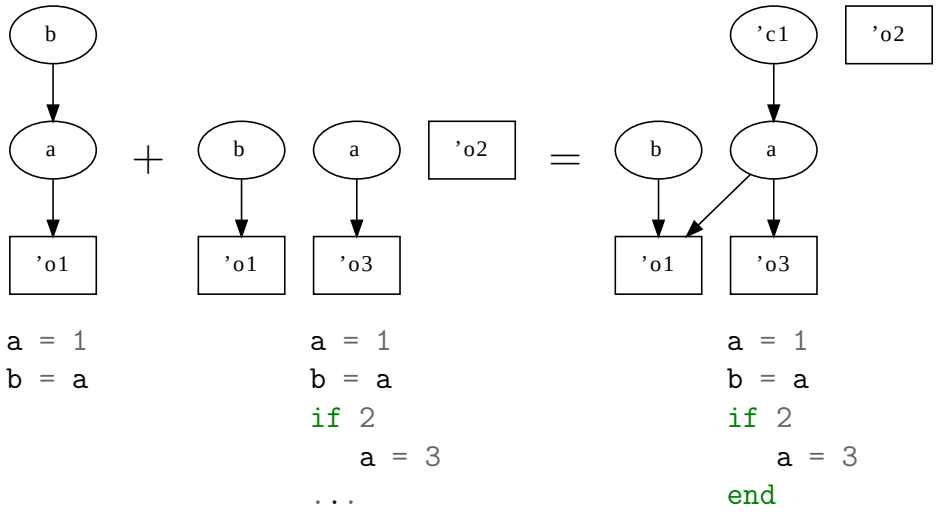

Figure 3.4: Merging graph built in a conditional block with the old connection graph.

The process of modelling a conditional block is depicted in Figure 3.4. The leftmost graph represents the situation before entering the conditional block. The graph in the middle is the situation after performing an assignment inside the conditional block. The rightmost graph is the new connection graph created through the merge and update procedure. Object node $'o2$ represents value 2 in the *if* statement condition. Reference node $'c1$ represents the value returned by the whole *if* statement.

Ruby *if* and *case* statements are expressions returning a value which also has to be modelled in a connection graph. The value of such statement is the value of the last expression from the chosen conditional branch. To reflect this, a special expression node $e \in V_{re}$ is created and linked with the nodes representing values of last expressions from each conditional branch.

Interprocedural analysis

Modelling of the method calls requires some more complex operations. Basically, it uses information from the connection graph of the callee is used to update the connection graph of the caller. To make the description easier to follow, I will introduce two auxiliary definitions.

Definition 6. $\textsc{PointsTo}(v)$ *is a set of all object nodes from* V_o *which are pointed to by reference node* $v \in V_r$.

$$o \in \textsc{PointsTo}(v) \Leftrightarrow$$
$$v \to o \in E_r \ \vee \ (\exists_{p \in V_r} \ v \to p \in E_d \ \wedge \ o \in \textsc{PointsTo}(p))$$

Intuitively, the PointsTo set is a set of all objects which can be reached from a given reference node following the reference or deferred edges. The

definition is recursive: if the connection graph contains cycles the recursion can be infinite. In practical implementation an additional mechanism is needed to avoid processing the same node twice.

Definition 7. *Consider a caller method A with connection graph CG^A and a callee method B with graph CG^B. Each subset from CG^A will be marked with ^{A} and each subset from CG^B will be marked with ^{B}.* $\textsc{MapsTo}(v)$ *is a set of all object nodes $o \in V_o^A$ from method A which correspond to object node $v \in V_o^B$ from method B.*

- *If $v \in V_{os}^B$:*
  - *If $v \in V_{os}^A$ it maps to itself.* $\textsc{MapsTo}(v) = \{v\}$
  - *Otherwise, it maps to a new object $n \in V_{os}^A$.* $\textsc{MapsTo}(v) = \{n\}$
- *If $v \in V_{og}^B$ it maps to itself (possibly initiating the creation of a new node $v \in V_{og}^A$)*
- *If $v \in V_{oph}^B$:*
  - *If $v \in V_{op}^B$ it maps to an actual object $a \in V_{os}^A$ which was passed as a corresponding parameter.* $\textsc{MapsTo}(v) = \{a\}$
  - *If $v \in V_{of}^B$ it maps to all objects from V_o^A pointed to by field with the same id of a corresponding object from A.*

$$o \in \textsc{MapsTo}(v) \Leftrightarrow \exists_{f \in V_{rf}^A} \textsc{Fid}(f) = \textsc{Fid}(v) \wedge$$
$$\textsc{Owner}(f) \in \textsc{MapsTo}(\textsc{Owner}(\textsc{Initiator}(v))) \wedge$$
$$o \in \textsc{PointsTo}(f)$$

The definition of the MapsTo set is quite complex. Generally speaking, the role of the set's construction procedure is to translate the names of the object nodes from one connection graph to another and to establish a relation between them. Some of its characteristics have to be emphasised.

First, all of the standard objects present in B and absent in A are mapped to new objects. This means that if B were a method creating an object and returning it as a return value, the objects returned by different calls of B would be treated as different objects. Two separate concrete objects can be modelled as a single abstract object in the B connection graph and as two abstract objects in the A connection graph. This is desirable behaviour because abstract objects should be distinguished if they have been created with separate program statements—from the perspective of method A such object creation statement would be a call of method B.

Second, a phantom field object from B can be mapped to multiple objects in A. This is because it represents all the values that an instance variable could have at the moment of the method call.

Lastly, in the case of a phantom field object the definition of MapsTo is recursive. It is, however, guaranteed that the recursion will stop at some point because the initiator of the first phantom field object created has to be a field node owned by object $o \in V_{op} \cup V_{os} \cup V_{og}$.

Declaring a method

```
def foo; end
```

For each method a separate connection graph is created. The special phantom object node *self*, representing an object being the method's owner, and special reference node *return*, pointing to a value returned from a method, are added. Each method parameter is modelled with a phantom object node representing the object passed to the method and a reference node pointing to that object node—this way the parameters can be treated exactly like local variables.

A special *return* node is linked with a node representing the value of any explicit *return* statement as well as with a node representing the value of the last statement in the method's body.

The initial escape state of the *self* node, parameter object nodes and the *return* node is set to *arg escape*. After the method's body is fully translated, escape analysis is performed as described before.

In Skarb all methods are translated on demand. The method declaration is analysed and the connection graph is built only when the method is actually called and not when the declaration is encountered.

Calling a method

```
foo(1,2,3)
```

Each method returns a value. To model it a special reference node is created. Then the caller graph is updated based on the callee graph. The nodes which can be affected by changes are nodes representing the object passed as parameters to the method and the node representing the returned value. The object being the callee owner is treated as an additional parameter corresponding to the *self* phantom node. By A I will denote the caller method, and by B the callee method.

For each pair (o_a, o_b), where $o_a \in V_o^A$ is a node representing an actual parameter and $o_b \in V_{op}^B$ is a node representing a formal parameter, the update procedure UpdateObjNode is performed. There could be multiple objects which can be passed as an actual parameter o_a. In that case multiple pairs have to be generated. Then, for pair (r_a, r_b), where $r_a \in V_{re}^A$ is the reference node created to model the returned value and $r_b \in V_{re}^B$ is the special *return* node of B method connection graph, update procedure UpdateRefNode is performed. After these two operations the caller graph is up to date.

Algorithm 3 UpdateObjNode procedure.

```
function UpdateObjNode(o_a, o_b)
    if Escape(o_b) = global escape then
        Set escape state of o_b to global escape.
    end if
    for all f ∈ V^B_rf : o_b → f ∈ E^B_f do
        Create node f ∈ V^A_rf if necessary.
        UpdateRefNode(f, f)
    end for
end function
```

The pseudocode for UpdateObjNode is presented in algorithm block 3. The procedure sets an escape state of the updated node to *global escape* if the corresponding node state was set to such. Let us notice that only the *global escape* state can be transferred from one method to another. The other escape states depend only upon the placement of node in method's connection graph. After the escape state is updated, the procedure assures the existence of all object fields and updates them.

Algorithm 4 UpdateRefNode procedure.

```
function UpdateRefNode(r_a, r_b)
    if ¬(∃_{p∈V^B_of} Initiator(p) = r_b ∧ p ∈ PointsTo(r_b) then
        Delete all outgoing edges of r_a.
    end if
    for all o ∈ PointsTo(r_b) do
        for all m ∈ MapsTo(o) do
            if m ∉ PointsTo(r_a) then
                Create node m ∈ V^A_o if necessary.
                Add edge r_a → m ∈ E^A_r.
            end if
            UpdateObjNode(m, o)
        end for
    end for
end function
```

The pseudocode for UpdateRefNode is presented in algorithm block 4. The first thing to do is to check whether the phantom object node representing the initial value of the field is still pointed to by the field reference. The lack of such an edge or the lack of a phantom node means that the previous value of the field was overwritten—this is modelled by deleting all of the outgoing edges of the reference node. Then it is necessary to update all of the objects pointed to by the reference by taking into account the rules by which the objects from one method map to objects from another method.

If the connection graph contains cycles, calling these procedures could lead to an infinite recursion. To prevent this an additional mechanism is needed to avoid processing the same pair of nodes twice. A simple solution is remembering all of the pairs processed so far and breaking the recursion if the same pair is encountered for the second time.

Recursive methods The recursive method is a method which at some point calls itself. To correctly model such a method a special strategy is needed. First, a connection graph is created for the method without evaluating recursive calls. Then the method is analysed for the second time and a new connection graph is built. When a recursive call is encountered the old connection graph is used as the graph of the called method. Since the old connection graph captures all of the non-recursive aspects of the method and in a correct program every recursion has to be finite, this approach results in the correct model.

The same approach can be used with indirect recursion. Generally, during the first pass calls to the methods present on the 'call stack' (or the static equivalent of it used during translation) will not be modelled. During the second pass incomplete connection graphs built during the first pass will be used.

Unknown methods As Ruby is a dynamically typed language, it is not always possible to determine which method should be called at compile time. Sometimes there is a need to generate a dynamic method call which will perform method search operations at run time. Such call should be represented in the caller's connection graph even though there is no connection graph of the callee. Such a case should be modelled in a conservative way: all objects passed as arguments should have a *global escape* state set, and the returned value should also be a new object also with a *global escape* state set.

3.2 Stack allocation

After the escape analysis is performed, it is possible to allocate all local objects on the stack instead of allocating them on the heap. This means changing the allocation call in the generated code. The benefits of stack allocation were described in the previous chapter. The practical implementation of this mechanism in the Skarb compiler showed some of its limitations.

With stack allocation instruction allocating the memory must appear in the same function in which that memory chunk will be used. Objects returned from method calls obviously have to be allocated on the heap. This means that it is impossible to use the factory method design pattern with stack allocation. Furthermore, only a basic object structure can be allocated on the stack. If an object needs additional memory and allocates it using its own internal methods, that memory has to be allocated on the heap. This is usually the case with collections and character strings of variable length.

Another problem is the risk of stack overflow caused by a large number of stack-allocated objects. Without providing some countermeasures against this problem, the stack allocation mechanism is not suitable for any practical use. I have applied two complementary solutions: a limit of bytes allocated on a single stack frame and a restriction on stack allocation inside a loop construct.

The limit concerning the stack frame works in fairly straightforward manner. In each translated method there is an additional counter tracking the number of bytes allocated on the stack at run time. After each allocation the counter is increased accordingly. When the value of the counter reaches a defined limit, all subsequent stack allocations are changed to heap allocations.

The drawback of the imposed limit is the necessity of checking and updating the value of the counter. These additional operations could make execution of the program significantly slower. To address this issue, I have decided to make the pre-conditions for stack allocation more restrictive: an object created inside a loop construct should never be allocated on the stack. This is based on the assumption that most of nontrivial loops is long enough to exceed the allocated bytes limit with objects created inside them. Whether this restriction leads to performance improvement or not was the subject of experimental tests.

The mechanism for reusing memory allocated for local objects, which is described in the next section, can also be seen as a form of countermeasure against the stack overflow problem, although it has a broader use.

3.3 Local objects reuse

The program heap memory is managed by the garbage collector. When a memory chunk becomes unreachable it can be reclaimed and reused for allocations of new objects. On the other hand, memory allocated on the stack frame is reclaimed all at once when the function exits, even if some parts of this memory became unreachable earlier. This can lead to a potential waste of memory and can limit the number of objects allocated on the stack. I propose a method based on the compile time garbage collection concept which allows to reuse some of the unreachable memory before the function exits.

The general idea is to use the connection graph to find local objects unreachable at some point of the execution and to reuse their memory for the newly created objects of the same type. Only local objects can be reused in such a manner because it is impossible to determine the reachability of a non-local object based on information from the connection graph. However, as long as the object does not escape from the method, it is irrelevant from the perspective of object reuse whether it is stack-allocated or heap-allocated. This means that the described optimisation can be applied independently of the stack allocation mechanism.

There are two kinds of expected benefits. First, when used in a conjunction with stack allocation, this optimisation should increase the number of objects allocated on the stack. Secondly, this way of obtaining memory for a created

object is much faster than heap allocation and even slightly faster than stack allocation, so changing the normal allocation instructions to object reuse should lead to an instant performance gain.

Additional information in a connection graph

To keep track of object reachability, information stored in the connection graph has to be extended. First, it should be noticed that objects created inside conditional blocks may or may not exist outside them—during static analysis it is impossible to say if the block was executed or not. To represent this in a connection graph the additional property of object nodes from V_{os} is introduced: existence state.

The existence of a node can be either *certain* or *conditional*. The initial existence state is always *certain*. When a connection graph of a conditional block is merged with the previous connection graph, the object nodes which exist only in the first graph have an existence state set to *conditional*. More formally, suppose that there is a connection graph representing a situation from before opening of the conditional block $CG^{'} = (V^{'}, E^{'})$. Analogously, a connection graph built in the conditional block is denoted by $CG^{''} = (V^{''}, E^{''})$. After the merge, each object node o such that $o \in V^{''}$ and $o \notin V^{'}$ has an existence state set to *conditional*. Obviously, only the memory of objects whose existence in a given block is *certain* can be reused.

Another aspect that is important from the perspective of reachability analysis is the treatment of objects being arguments of evaluated expression. Until the expression is evaluated, those objects stay reachable even if they are not pointed to by any explicit references. To assure this along each connection graph a special data structure is stored—the expression stack.

The expression stack consists of expression nodes belonging to V_{re} and representing all evaluated expressions. Opening an expression results in pushing a new node to the top of the stack, while closing an expression results in removing the top node. For each node $v \in V$ representing an argument of an expression, there exists an edge $e \rightarrow v \in E$, where $e \in V_{re}$ is a node representing the expression. Removing an expression node from the stack leads to deleting all of its outgoing edges. This way it is guaranteed that during the expression evaluation all of its arguments are pointed to by at least one reference node.

Special attention has to be paid to the modelling assignments. It is important that the BYPASS operation on the reference node be performed after the arguments of the right-hand expression are evaluated but before the evaluation of the whole expression. This allows for a situation where an object being the old value of the variable is reused to store the new value.

Finding objects for reuse

When a statement leading to object creation is encountered, the search for currently unreachable objects is performed. I will denote a created object by o.

From the set of all unreachable objects those with a *certain* existence state and type compatible with o are selected and stored alongside the o node. This set comprises objects whose memory can be reused by o. Until the method is fully translated the connection graph is incomplete, thus it is impossible to perform escape analysis and a stored set may contain non-local objects. They will be rejected later.

Definition 8. PotentialPrecursors(o) *is a set of objects whose memory can be potentially reused by* o*.*

To find unreachable objects a simple mark-and-sweep algorithm is used. The root set contains nodes representing local references, method parameters, global variables and nodes from the expression stack. The connection graph is traversed starting from the root set and all encountered objects are marked. Unmarked objects are treated as unreachable.

Constructing succession lines

When a method is fully translated and its connection graph is completed, an escape analysis is performed. At that point set of local objects can be obtained. For each of these objects a PotentialPrecursors set exists. The remaining problem is to decide in what manner the memory will be reused. I will define it formally:

Definition 9. *Succession line is a totally ordered set of local objects* $(o_1, \ldots, o_n)$ *such that:*

$$\forall_{i \in \mathbb{N}, 1 \leq i < n} \;\; o_i \in \text{PotentialPrecursors}(o_{i+1})$$

In a succession line only the memory for the first object is allocated normally, the memory for all subsequent objects is reused. The goal of an algorithm solving the memory reusage problem is to create the smallest number of succession lines which will work as the partition of the local objects set.

My initial approach to this problem was the full search of a solution space with the simple pruning of branches where the number of succession lines exceeded the current minimum. The experiments quickly proved that such a search is too slow for any practical use. Therefore, I decided to apply the form of a greedy algorithm. It can be easily shown that it is not optimal, nevertheless, in most cases it leads to a good enough solution.

Definition 10. PotentialSuccessors(o) *is a set of local objects such that for any local object* p*:*

$$p \in \text{PotentialSuccessors}(o) \Leftrightarrow o \in \text{PotentialPrecursors}(p)$$

The pseudocode for the greedy algorithm for the construction of succession lines is presented in algorithm block 5. It attempts to always choose an object with the highest number of potential successors to maximise the probability

Algorithm 5 Greedy algorithm for constructing succession lines.

$O \leftarrow$ local objects
$S \leftarrow$ new succession line
Find object $o \in O$ with the biggest $O \cap \textsc{PotentialSuccessors}(o)$ set.
Add o to S.
Remove o from O.
while $O \neq \emptyset$ **do**
    $s \leftarrow$ last object from S.
    Find object $o \in O \cap \textsc{PotentialSuccessors}(s)$ with the biggest $O \cap \textsc{PotentialSuccessors}(o)$ set.
    if no o found **then**
        $S \leftarrow$ new succession line
        Find object $o \in O$ with the biggest $O \cap \textsc{PotentialSuccessors}(o)$ set.
    end if
    Add o to S.
    Remove o from O.
end while

that the succession line will continue. If it is impossible to find the next object for the succession line, a new line is started. The algorithm stops when all local objects are distributed between succession lines.

Chapter 4

Tests results

To verify if the proposed optimisations are effective and if they introduce expected performance gains, several tests were performed. The test cases consisted of simple programs from an older version of the Computer Language Benchmarks Game [10]:

- Ackermann—calculating value of the Ackermann function,
- Matrix—multiplication of large matrices containing floating-point numbers,
- Quicksort—sorting large arrays of floating-point numbers with Quicksort,
- Spectral norm—calculating the spectral norm of a large matrix of ones.
- Binary trees—allocating and deallocating a large number of binary trees.
- Binary trees (small)—identical to the previous test but with smaller tree depth.

Different types of metrics were measured: execution time, percentage of time spent on garbage collection and percentage of concrete objects in each program that can be either stack-allocated or reused for later allocations.

The execution time of the same program in a modern operating system can vary due to various factors (including garbage collection, which is not fully deterministic). To collect more reliable results, each program was executed ten times, the biggest and the smallest results were discarded and from the rest mean execution time was calculated.

The fraction of time spent on garbage collection was measured using gperftools [11] profiler. Each program was profiled five times in a row and a mean value was calculated.

To measure the number of allocated objects a special version of Skarb was prepared. Programs compiled with it keep two separate counters for heap-allocated and stack-allocated objects. The counters are updated during program execution and the values are printed when the program exits. In this case, the number of allocated objects was constant for a program, so only one run was needed to collect the exact values.

All test programs were compiled with math inline optimisation, which ensures that compound arithmetical expressions are translated to standard C operators instead of function calls. The C source code generated by Skarb was compiled with a gcc 4.7 compiler with **O3** level of optimization enabled. Programs were executed on computer with AMD Phenom II X2 555 3.2 GHz processor, the operating system was 64-bit version of Arch Linux.

Stack allocation in loop constructs

The goal of the first test was to verify if the restriction on stack allocation inside the loop constructs is needed and if it performs better than just a standard limit on bytes allocated on the single stack frame.

	Skarb (SA)		Skarb (SANL)	
	m	s	m	s
Ackermann	0.350	0.004	0.351	0.002
Matrix	2.069	0.226	2.060	0.230
Quicksort	2.668	0.068	2.648	0.084
Spectral norm	0.870	0.041	1.074	0.037
Binary trees	12.500	0.274	12.473	0.112

Table 4.1: Execution time of programs with restriction on stack allocation inside loops (SANL) and without such restriction (SA). m denotes mean time, s is the standard deviation. All times are given in seconds.

The results are presented in Table 4.1. Column 'Skarb (SA)' contains the execution times of programs with a standard bytes limit, column 'Skarb (SANL)' contains the execution times of programs with a standard bytes limit and a restriction on allocation inside the loops. As one can see, no significant improvement is introduced by loop allocation restriction. In case of spectral norm calculation it even slows down the program. It turns out that the elimination of additional byte counter operations is not important for the whole program execution. In consequence, in further tests only a standard byte limit was used.

Percentage of optimised objects

Before measuring the execution time of individual test programs, it would be interesting to know what level of performance gain could be expected in each case. Such a prediction can be made based on the number of object which were subject to optimisation. Two versions of each program were compiled: with stack allocation optimisation and with object reuse optimisation. The numbers of stack-allocated and heap-allocated objects were collected during a single program run. The number of objects whose memory was reused can be

calculated as the difference between the total allocated objects number in the two versions of the program.

	SA%	OR%
Ackermann	67%	0%
Matrix	0.00004%	1%
Quicksort	4%	30%
Spectral norm	0.0003%	0.0002%
Binary trees	20%	0%

Table 4.2: Percentage of total concrete objects than can be stack-allocated (SA) or reused (OR).

The results are presented in Table 4.2 as the percentage of all created objects. In the case of the recursive Ackermann function more than half of all objects can be allocated on the stack, which should result in substantial performance gain. In the binary trees test 20% of the objects can be stack-allocated, so a noticeable speedup is also expected. The same goes for quicksort, where performance gain is expected due to high percentage of reused objects. Based on this data, no speedup should be expected in the matrix multiplication and spectral norm tests.

The high percentage of objects that can be stack-allocated in the Ackermann and binary trees test programs stems from their recursive character and lack of loops. Furthermore, in the Ackermann function no arrays or other data structures are used, so there are no objects with internal heap-allocated data. Because of these reasons performance gain is expected to be most evident in the Ackermann test program.

Percentage of time spent on garbage collection

It is possible to do tests similar to those described in Chapter I to check how much of the program execution time is spent on garbage collection in optimised and non-optimised programs. Successful optimisation should reduce this value.

The values collected using the gperftools profiler are presented in Table 4.3. As expected, garbage collection overhead in the optimized Ackermann program is much lower than in the non-optimised version. There are no expected differences between the different versions of the binary trees program and the difference in the case of quicksort is rather small.

The results of the spectral norm and matrix multiplication tests are surprising. Intuitively, based on the fraction of objects affected by optimisations, no difference in garbage collection times was expected. However, a small reduction is seen in the spectral norm test and a small gain in the matrix multiplication. This can be explained by complex garbage collector strategies and different object access time depending on their placement in the memory. This way

	Skarb	Skarb (SA)	Skarb (OR)	Skarb (SA + OR)
Ackermann	55.3%	33.7%	60.4%	22.7%
Matrix	31.5%	34.3%	39.9%	39.5%
Quicksort	55.4%	54.8%	51.8%	49.4%
Spectral norm	45.2%	41.0%	39.8%	41.2%
Binary trees	56.4%	55.1%	54.6%	55.1%

Table 4.3: Percentage of execution time spent on heap memory management in programs compiled with different optimisations: SA—stack allocation, OR—object reuse, SA+OR—both.

even a few objects allocated on the stack or not allocated at all can change the moment when a garbage collection cycle occurs. Similarly, moving objects in the memory can change the time of execution of code, thus leading to an increase in the portion of time spent on memory management.

Performance gains

In the final experiment the mean times of execution of all the test programs were collected. Because the previous tests showed that in every case the fraction of time spent on garbage collection changes, the performance gain or reduction can be expected in any program.

	Skarb		Skarb (SA)		Skarb (OR)		Skarb (SA+OR)	
	m	s	m	s	m	s	m	s
Ackermann	0.566	0.009	0.350	0.004	0.563	0.010	0.359	0.011
Matrix	2.067	0.225	2.069	0.226	1.775	0.193	1.783	0.184
Quicksort	2.566	0.078	2.668	0.068	2.278	0.064	2.330	0.071
Spectral norm	0.958	0.022	0.870	0.041	0.884	0.031	0.857	0.086
Binary trees	13.044	0.132	12.500	0.274	13.117	0.200	12.390	0.314
Binary trees (small)	2.738	0.014	2.573	0.023	2.719	0.020	2.567	0.018

Table 4.4: Execution time of programs compiled with different optimisations: SA—stack allocation, OR—object reuse, SA+OR—both. m denotes the mean time, s denotes the standard deviation. All times are given in seconds.

The results where gathered in Table 4.4. To make the gains introduced by each optimisation more visible, the execution times of optimised programs relative to the non-optimised ones were presented in Table 4.5.

As one can see, stack allocation optimisation introduces a huge gain in the case of Ackermann function and only a minor gain in the case of binary trees. This holds for both versions of binary trees tests with different tree depth level, so this behaviour seems independent of the amount of memory used or of the

	Skarb (SA)	Skarb (OR)	Skarb (SA+OR)
Ackermann	0.62	0.99	0.63
Matrix	1.00	0.86	0.86
Quicksort	1.04	0.88	0.91
Spectral norm	0.91	0.92	0.89
Binary trees	0.96	1.01	0.95
Binary trees (small)	0.94	0.99	0.94

Table 4.5: Execution time relative to a non-optimised program.

total execution time. A 10% speedup in the spectral norm test can be explained with a small change in the conditions, which delays the garbage collector cycle.

Surprisingly, object reuse optimisation proved to be the most effective in the matrix multiplication test introducing 15% gain. Since it reduces the execution time of the program while at the same time increasing the fraction of time spent on memory management, the speedup has to be connected with execution of a non-garbage collector code. Perhaps this can be explained with different objects placement in the memory or some optimizations made by gcc. In the spectral norm test a 10% speedup was observed, just as with the stack allocation optimisation. These gains, however, were not cumulative. A 10% gain quicksort test was expected based on the results of the previous experiments.

Chapter 5

Conclusions

In this thesis memory management mechanisms used in official implementations of popular script languages were identified and analysed. In the case of Perl this was reference counting, in the case of Python this was the hybrid mechanism using reference counting and a garbage collector and in the case of Ruby this was the standard garbage collector. Tests showed that memory management accounts for a large part of the execution time of programs written in these languages. One of the reasons for this situation is the fact that in the analysed languages there is no distinction between scalar values and compound objects. Even simple numerical values are often heap-allocated. This makes all kinds of memory management optimisations particularly useful in script language interpreters.

Two forms of optimisations were implemented for Skarb—experimental Ruby compiler: stack allocation and local object reuse. Both relied on static code analysis performed with the use of connection graph abstraction. This abstraction proved to be a convenient way of tracking the escape state of objects across the methods and it could be easily extended to keep additional information.

The applications of stack allocation were limited because of the requirement to allocate the memory on the topmost stack frame. This is this technique's natural restriction but makes it inflexible in practical use. Stack allocation was not used with objects collections because it would require modification of external libraries. I conclude that implementing stack allocation in the case of object oriented languages with code encapsulation features is quite inconvenient and not fully effective. More flexible mechanisms should be researched.

The experimental tests showed that the applicability of stack allocation and object reuse vary greatly between the programs. In the best cases 67% of the objects could be stack-allocated and 30% could be reused. However, even a small number of objects not allocated on the heap can improve program performance by improving the memory locality and affecting garbage collector cycles. The biggest speedup of program execution was 38% in the case of stack allocation and 14% in the case of object reuse.

The collected results are not representative of the whole range of programs

written in script languages. However, they clearly show that the proposed optimisations are of practical use and can introduce noticeable performance gains. Research on memory management strategies and optimisations has traditionally been performed for languages like Lisp, Prolog, Smalltalk and, more recently, for Java and C#. It is now time to move this research to the field of script languages—languages which would greatly benefit from this type of research.

Possible further research

The presence of unknown methods reduces the precision of escape analysis and consequently reduces the applicability of optimisations based on the connection graph. This means that the effectiveness of the described optimisations may be limited by the effectiveness of the type inference system, so improving the latter would also improve the former. In fact, since a connection graph allows to track program objects and to annotate them with any properties, it would be possible to build a type inference system relying on a connection graph itself. This would lead to a more precise analysis than the approach employed currentyl in Skarb. With some additional rules the types of objects contained in collections, such as arrays, could also be determined.

Replacing the stack allocation with a more flexible allocation strategy based on memory regions is a logical course of action. This would allow to maintain the benefits of fast allocation and deallocation and to extend them to a wider group of objects (e.g., objects returned from the factory methods).

Finding a better algorithm for building succession lines would improve the applicability of object reuse. Switching from tracking object reachability to tracking the moment of its last usage would allow to determine its lifetime more precisely. Strategies for reusing objects across methods could also be researched and developed.

Finally, it would be valuable to collect more experimental data. Testing implemented optimisations along different garbage collectors would allow to better assess their versatility. Checking their impact on the program's memory consumption would also be interesting.

Bibliography

[1] Emery D. Berger. Reconsidering custom memory allocation. In *In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2002*, pages 1–12. ACM Press, 2002.

[2] Bruno Blanchet. Escape analysis for Java: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[3] Hans J. Boehm, Alan Demers, and Mark Weiser. Boehm-Demers-Weiser garbage collector [online]. URL: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/` [cited 5.05.2012].

[4] Dov Bulka and David Mayhew. *Efficient C++: performance programming techniques*. Addison-Wesley, Boston, MA, USA, 2000.

[5] Craig Chambers. Cost of garbage collection in the SELF system. In *OOPSLA'91 GC Workshop*, 1991.

[6] Jong deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25:2003, 2003.

[7] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice & Experience*, 24(6):527–542, 1994.

[8] Amer Diwan, David Tarditi, and Eliot Moss. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems*, 13:244–273, 1995.

[9] Python Software Foundation. Python official website [online]. URL: `http://www.python.org` [cited 11.05.2012].

[10] Brent Fulgham. Computer language benchmarks game [online]. URL: `http://shootout.alioth.debian.org/` [cited 07.05.2012].

[11] Google. gperftools [online]. URL: `http://code.google.com/p/gperftools/` [cited 5.05.2012].

[12] Aman Gupta. perftools.rb [online]. URL: `https://github.com/tmm1/perftools.rb` [cited 5.05.2012].

[13] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Trans. Program. Lang. Syst.*, 10(4):555–578, 1988.

[14] Gerda Janssens, Maurice Bruynooghe, Nancy Mazur, and Peter Ross. Practical aspects for a working compile time garbage collection system for Mercury, 2001.

[15] Richard E. Jones and Rafael Dueire Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

[16] Wen ke Chen, Sanjay Bhansali, Trishul Chilimbi, Xiaofeng Gao, and Weihaw Chuang. Profile-guided proactive garbage collection for locality optimization. In *In Proceedings of ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 332–340, 2006.

[17] Jeffrey Kegler. Perl is undecidable. *The Perl Review*, 5(0):7–11, 2008.

[18] Feliks Kluzniak. Compile time garbage collection for ground Prolog. In *ICLP/SLP*, pages 1490–1505, 1988.

[19] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE '05, pages 111–120, New York, NY, USA, 2005. ACM.

[20] Doug Lea. A memory allocator [online]. URL: `http://gee.cs.oswego.edu/dl/html/malloc.html` [cited 5.05.2012].

[21] Henry Lieberman, Carl Hewitt, and Danny Hillis. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26:419–429, 1983.

[22] Nancy Mazur. *Compile-time garbage collection for the declarative language Mercury*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, May 2004.

[23] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. In *In Symposium on Programming Language Implementation and Logic Programming (PLILP'95)*, pages 241–258. Springer, 1995.

[24] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. *SIGPLAN Not.*, 27(7):116–127, July 1992.

[25] Perl.org. Perl official website [online]. URL: `http://www.perl.org` [cited 11.05.2012].

[26] Carl Mcconnell Ralph and Ralph E. Johnson. Compile-time garbage collection in typed Smalltalk.

[27] Vimal K. Reddy, Richard K. Sawyer, and Edward F. Gehringer. Caching strategies for improving generational garbage collection in smalltalk. Technical report, North Carolina State University, 2003.

[28] Patrick Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *In Functional Programming Languages and Computer Architecture*, pages 106–116. ACM Press, 1993.

[29] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, 1975.

[30] Jan Stępień and Julian Zubek. Translation of ruby source code to a language compilable to machine code. Bachelor thesis, Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland, 2011.

[31] Ruby Visual Identity Team. Ruby official website [online]. URL: `http://www.ruby-lang.org` [cited 11.05.2012].

[32] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, 1984.

[33] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, pages 1–116. Springer-Verlag, 1995.

Warszawa, dnia _____

Oświadczenie

Oświadczam, że pracę magisterską pod tytułem „Zastosowanie optymalizacji pamięci w interpreterach języków skryptowych”, której promotorem jest dr inż. Krzysztof Kaczmarski, wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

_____