Gulenko,
Allocation Removal in Modern Object Oriented VMs

Allocation Removal in Modern Object Oriented VMs

by

Anton Gulenko

A thesis submitted to the
Hasso-Plattner-Institute for Software Systems Engineering
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in Software Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Tim Felgentreff

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

December 5, 2014

Abstract

Modern programming languages abstract from many details of the underlying operating system and hardware. This allows programmers to focus on the structure of the program and write modular, expressive and reusable code. Unfortunately, these layers of abstraction also create a wide semantical gap between the execution environment and the underlying platform. The semantical difference has a big impact on the performance of dynamic programming languages. Today, most wide-spread virtual machines include just-in-time compilation to achieve usable performance. A key goal of JITs is to remove the abstractions at runtime by applying optimizations before generating the machine code.

Just-in-time compilation is a mature technology and frameworks such as RPython and Graal/Truffle allow to automatically add JIT support to VMs for different languages. This greatly reduces the complexity of VM implementations. While JITs perform many important optimizations automatically, some still have to be implemented by the VM itself. One example is the recent optimization technique "storage strategies", which helps the VM to save memory and increase execution speed. Most RPython VMs include a custom implementation of this optimization, which leads to code duplication and increased code complexity.

We show that it is possible to extract language-independent aspects of storage strategies and present rstrategies, a reusable implementation for RPython based VMs. Experiments show that the rstrategies library delivers good speedups in favorable situations, while never resulting in a severe performance loss.

Zusammenfassung

Moderne Programmiersprachen abstrahieren von vielen Details der unterliegenden Betriebssysteme und Hardware. Dies erlaubt es Programmierern, sich auf die Struktur von Programmen zu konzentrieren und modularen, ausdrucksstarken und wiederverwendbaren Code zu schreiben. Leider führen diese Abstraktionsschichten auch zu einem großen semantischen Unterschied zwischen der Ausführungsumgebung und der unterliegenden Plattform. Dieser semantische Unterschied hat großen Einfluss auf die Performance von dynamischen Programmiersprachen. Heutzutage enthalten die meisten verbreiteten virtuellen Maschinen just-in-time Compiler, um eine gute Performance zu erreichen. Eine wichtige Aufgabe von JITs ist es, die Abstraktionen zur Laufzeit zu entfernen, indem sie den generierten Maschinencode optimieren.

Just-in-time Kompilierung ist eine ausgereifte Technologie und Frameworks wie RPython und Graal/Truffel erlauben die automatische Generierung von JIT-Unterstützung für VMs verschiedener Programmiersprachen. Dies reduziert die Komplexität von VM Implementierungen. Obwohl JITs viele wichtige Optimierungen automatisch durchführen, müssen einige dennoch von der VM selbst implementiert werden. Ein Beispiel hierfür ist die neue Optimierung “storage strategies”, die der VM hilft Speicher zu sprachen und die Ausführungsgeschwindigkeit zu erhöhen. Die meisten RPython VMs enthalten eine eigene Version dieser Optimierung, was zu Code-Duplizierung und erhöhter Code-Komplexität führt.

Wir zeigen, dass es möglich ist sprachunabhängige Aspekte von “storage strategies” zu extrahieren und präsentieren rstrategies, eine wiederverwendbare Implementierung für RPython basierte VMs. Experimente zeigen, dass die rstrategies Bibliothek in günstigen Situationen gute Performancegewinne liefert während ernste Performanceeinbrüche vermieden werden.

Acknowledgments

I want to express my gratitude to Robert Hirschfeld and Tim Felgentreff for the opportunity to work on this interesting topic. Thank you for countless fruitful discussions.

Many thanks to Carl Friedrich Bolz for valuable feedback.

Further, I want to thank Tobias Pape for his great LaTeXproficiency.

And finally I want to thank my family Natalya, Maryna, Dieter, Janine and my friends Tobi and Niko for constantly supporting me and pushing me forward!

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

JIT	just-in-time compiler
VM	virtual machine
AST	abstract syntax tree
API	application programming interface
IR	intermediate representation

1 Introduction

Modern programming languages are designed with dynamic features such as polymorphism, dynamic typing or late class binding. Such features create layers of abstraction between the language semantics and the physical machine they are executed on. These abstractions help programmers write more concise and powerful programs by providing concepts of modularity and code reuse. However, the big abstraction gap can also have a second effect, which is the necessity for complex runtimes and virtual machines (VMS).

1.1 Virtual Machines

A VM forms a layer on top of the physical machine and operating system. It receives a *user program* as input and executes it, acting as an interface to the underlying layers. Using a VM is an alternative approach to directly compiling the user program to machine code. Direct compilation of dynamic programming languages is rarely possible with satisfying performance results, due to the said abstraction gap.

The core element of every VM is the execution engine. The execution engine takes a representation of the user program and executes it by interpreting a stream of instructions. It is also called the *interpreter*. During the process of interpretation, it interacts with all the other elements of the VM.

The main property of the execution engine is the type of user program representation it can interpret. There are two main distinctions. An *abstract syntax tree (*ast*) interpreter* takes the AST of the user program as input and interprets it by navigating its nodes. The AST is usually represented as an internal data structure. A *bytecode interpreter* on the other hand, consists of an endless loop interpreting a stream of bytecode instructions. This is very similar to the fetch-decode-execute mechanism of a physical processor.

Regardless of the kind of execution engine, the VM must first convert the source code of the user program into the appropriate representation. This can be done either by a parser component at VM startup, or by a separate compiler program that is also part of the programming language infrastructure.

1.2 Tracing JITs

A simple execution engine alone is not enough to fulfill the requirements of modern VMS. Programmers also rely on VMS to execute their dynamic code with sufficient performance. Therefore VMS must implement various optimization techniques. An important part of the modern optimization landscape are just-in-time compilers (JITS). JITS monitor the execution of their underlying program and produce optimized machine code during runtime. A *method* JIT optimizes the user program on method level, often making assumptions about the types of parameters and variables to enable better optimizations. *Tracing* JITS, on the other hand, identify entire loops in the user program. If a frequently executed loop is encountered, it is optimized by collecting every operation in the loop into a one-dimensional trace. That trace is then used to produce optimized machine code for an entire loop iteration.

There are multiple implementations of tracing JITS. TraceMonkey [12] is a JavaScript VM featuring a tracing JIT. The RPython toolchain [1] is designed for developing VMS using a subset of Python. A powerful tracing JIT can be generated automatically [4].

Tracing JITS have access to an entire sequence of language operations which can span over many methods, objects and classes. This information can be used to implement powerful and aggressive optimizations, resulting in very specialized machine code for the current runtime situation. Basic optimizations include inlining, dead code elimination, method splitting and specialization. One of the most powerful techniques is escape analysis, which enables allocation removal optimizations.

1.3 Allocation Removal in RPython

RPython allows the automatic generation of an optimizing tracing JIT. The generated JIT and the RPython toolchain implement most typical JIT optimizations. In RPythons tracing JIT one of the main performance optimizations is allocation removal.

Allocation removal [3] is a class of optimizations that try to avoid memory allocations on the heap requested by the user program while still preserving correct program semantics.

The allocation removal implemented by RPythons tracing JIT is called static objects. It identifies objects that never leave the current trace and avoids allocating them; their contents are maintained on the stack instead.

Another allocation removal optimization is called **storage strategies**. Storage strategies try to optimize collection datatypes by transparently inlining their elements data instead of holding references to actual objects. Object allocations can be avoided because memory for the objects is already reserved in the collection itself. In contrast to RPythons static objects, this optimization is applicable to any object, not only those that don't escape the loop that is being optimized by the JIT.

Certain conditions have to be met in order to apply storage strategies. These conditions are hard to detect automatically even with the rich information available to a tracing JIT. Therefore storage strategies have been implemented for PyPy [5], an RPython based Python interpreter, but are not automatically provided by the RPython toolchain.

Bolz et al. have shown that storage strategies are beneficial for the Python programming language [5]. Since Python offers one generic list datatype, optimizing it can have a big impact on program performance. We argue that Bolz et al.'s approach can be generalized and applied to other programming languages like Smalltalk or Ruby, which have rich collection standard libraries.

In this thesis we show how we can make the storage strategy approach generic and gain its performance benefits in other contexts than the Python programming language. We present the **rstrategies** library – a reusable library for RPython based VMS to implement the storage strategies optimization. Our experiments indicate that **rstrategies** can increase the performance of allocation-heavy programs by up to 66%, while keeping the potential performance overhead in the 1-2% range.

The main contributions made by this thesis are the following:

1. An overview of RPythons tracing JIT, allocation removal techniques and storage strategies in particular.
2. The design and implementation of a reusable, VM-independent RPython library for storage strategy optimizations.
3. Evaluation of the presented library through performance measurements.

1.4 Thesis Structure

The rest of this thesis is organized as follows.

The first part of chapter 2 introduces the RPython toolchain. It explains why a toolchain like RPython is useful for programming language research and provides guidelines on how to implement VMS. The second part of chapter 2 discusses allocation removal optimizations. In particular, their applicability in RPython VMS and expected performance benefits are highlighted.

Chapter 3 describes the design of the **rstrategies** library for implementing storage strategies. After introducing the main library parts, we explain the design decisions and the scope of the library.

Chapter 4 dives into implementation details of the **rstrategies** library. We provide code examples both of the library code and of the VM code using **rstrategies**.

Chapter 5 evaluates **rstrategies** from two perspectives. First, we provide performance measurements of the implemented optimizations by comparing different benchmarks executed with and without storage strategies. Second, we show the reusability of the library across different language implementations by listing several VMS that have been refactored in order to use **rstrategies**.

Chapter 6 presents related work in the field of modern, optimizing language implementations, and allocation removal techniques in particular.

Chapter 7 concludes the thesis by summarizing our results.

2 Allocation Removal Techniques

2.1 The RPython Toolchain

Language designers research novel programming language concepts and develop new ways to express program behavior and structure. Although language design includes a theoretical component, the researchers need to test their ideas in practice, which requires the construction of prototypes and experiments. In other words, language designers have to implement programming languages as quickly as possible.

Implementing a programming language from scratch is difficult and requires at least either a static compiler or a VM. It is not enough to produce a naive and simple implementation since that would render the prototype unusable in many real-world scenarios, making it impossible to thoroughly evaluate the new concepts. Even a prototype language implementation has to provide appropriate performance. On the other hand, going through the entire process of creating a high-performance compiler or VM over and over is barely feasible. Luckily, it is possible to create reusable frameworks and toolchains for exactly the purpose of implementing a programming language.

In the world of statically compiled languages, compiler frameworks have been around since the late 1980s. The Gnu Compiler Collection[1] is a widely used compiler framework, with front ends for dozens of programming languages (though not all are part of the official release) and even more back ends. LLVM[2] is a more recent example of a compiler infrastructure aiming for better modularity [19]. Compiler frameworks typically define a three-tier architecture: a front end parses the program source code into an intermediate representation, which is then optimized before being lowered to machine code using one of the supported back ends. By using the intermediate representation as an interface between front ends and back ends, the programming language parsing can be nicely separated from details of machine code, thereby greatly reducing overall complexity. In order to implement a compiler, a language designer only needs to provide a front end.

[1] `https://gcc.gnu.org/` (August 2014)
[2] `http://llvm.org/` (August 2014)

As great as compiler frameworks are, they are not enough to fully implement modern dynamic languages. Such languages often require at least a garbage collector component, and a high-performance VM needs some kind of JIT to deliver good performance. Both of these components are very complex, which makes it impractical to implement them separately for every VM project.

The RPython Toolchain [1] aims to help programmers experiment with VMS for dynamic programming languages. The toolchain takes a lot of implementation work, typically required in order to implement a VM, out of the programmers hands.

There are two important aspects to the toolchain: the RPython language and the automatic translation process. VMS are written in the RPython language, which is a restricted subset of Python. The two main restrictions of RPython are static typing and a static layout of classes and methods. After an initialization phase, the RPython program is "frozen" and further modifications to the program structure are not possible. The program can still take advantage of Python features such as polymorphism, mixins and first-class methods. Also, a VM written in RPython can still be executed by a regular Python interpreter. Though this will not achieve great performance, it is a very important feature of RPython since it allows to naturally debug and test a VM like any other Python program. The translation process is essential for the performance of an RPython VM: it translates the VM code into efficient C code. Due to the static typing the compiler can conduct whole-program analysis and already perform certain optimizations like inlining or method specialization. Furthermore, the translation process automatically adds two important modules to the VM:

GARBAGE COLLECTOR RPython is a fully garbage collected language. A VM written in RPython can take full advantage of this, completely eliminating the need to explicitly deal with memory management. There is a variety of garbage collector implementations to choose from.

TRACING JIT COMPILER The toolchain can construct a tracing JIT compiler, specialized for the implemented VM. The JIT is completely independent of the language or program executed by the VM and is the main driver of performance.

2.1.1 Building VMs with RPython

Many languages have been implemented using the RPython toolchain, including the Python VM PyPy [21], the Ruby VM Topaz [10] and the Squeak VM RSqueak [6]. Although the RPython toolchain generates certain parts of the VM automatically, all of these projects exhibit certain patterns regarding their architecture. Figure 2.1 shows the high level architecture typically found in

RPython VMS. A parser component transforms programming language artifacts (usually source code) into an internal representation, which is then interpreted by the execution engine. The typical choices for the internal representation are ASTS or bytecodes; the execution engine is implemented to handle the chosen representation.

The *object model* is a hierarchy of RPython classes representing all primitive types and data structures defined in the programming language. All parts of the object model must be derived from a single top-level class in order to enable polymorphic use of their instances in RPython. Figure 2.2 shows a part of the object model in the RSqueak VM. The classes `W_SmallInteger` and `W_Float` represent primitive numerical datatypes, the rest of the hierarchy represents different kinds of objects. Although instances of Squeaks `SmallInteger` and `Float` classes act like regular objects, they are still special enough to be treated separately by the VM. Instances of `W_SmallInteger` and `W_Float` are examples of *boxed objects*: both classes contain a single value of RPythons `int` and `float` type, respectively. Boxed objects is an important concept in RPython VMS. Classes like `W_Float` are required to take advantage of RPythons dynamic features, but the actual operations still happen on the *unboxed* values.

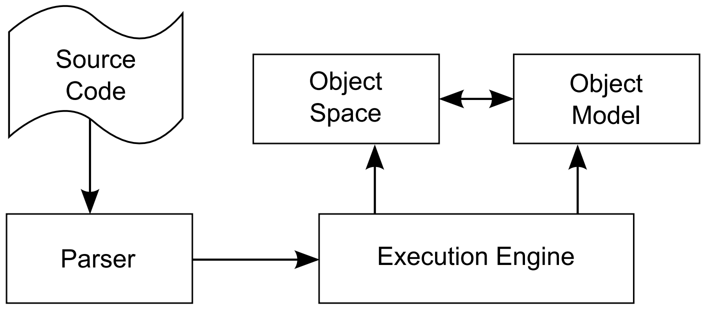

Figure 2.1: High level components of a typical VM written in RPython

The object space provides an interface for creating and managing objects. Amongst other things, it offers routines for boxing and unboxing of boxed objects in the VM. Through this abstraction it is possible to make experiments with the object layout and representation, without affecting the rest of the VM code.

2.1.2 RPythons Tracing JIT

The RPython toolchain can automatically add a meta tracing JIT compiler [4] to the VM. A tracing JIT compiler works under the assumption that the

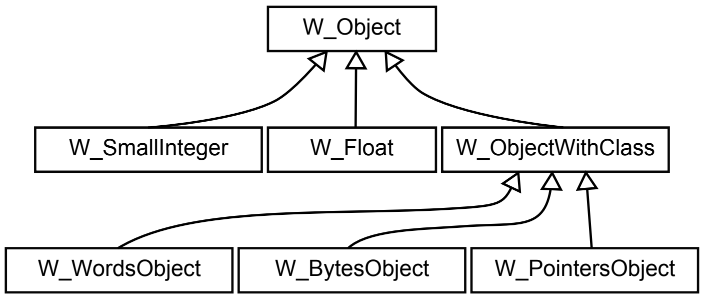

Figure 2.2: Excerpt of RSqueaks object model

optimized program spends most of its time in loops, and that it will most likely follow the same code path on each iteration. The tracing JIT detects loops at runtime and appends execution counters to them. At first, the loops are executed normally. When a loop counter exceeds a certain threshold, it is considered a *hot* loop and the JIT compiler will trace the next iteration. Tracing is a separate mode of execution, in which every operation that is performed is collected into a flat list. The resulting *trace* is rich of contextual information. The JIT then proceeds to optimize the trace and produce native machine code. The next loop iteration is executed by jumping directly into the optimized machine code. Figure 2.3 summarizes the different execution modes for a given loop and the events that lead the VM to switch between these modes.

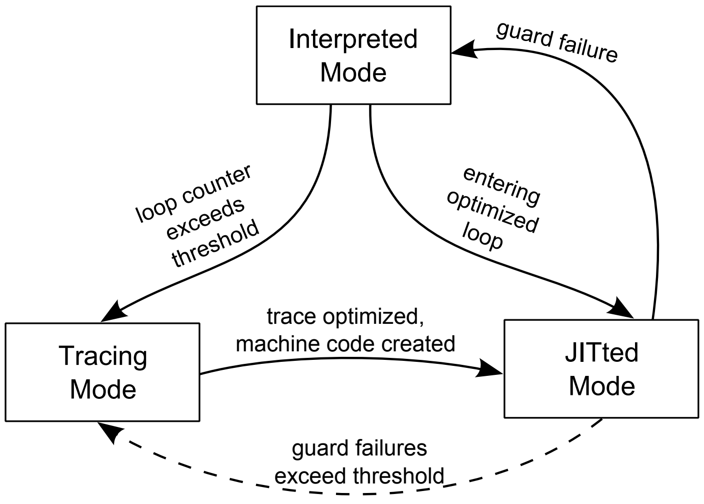
Interpreted Mode
guard failure
loop counter exceeds threshold
entering optimized loop
Tracing Mode
trace optimized, machine code created
JITted Mode
guard failures exceed threshold

Figure 2.3: Execution modes of a VM with a tracing JIT compiler

Listing 2.1 shows a simple `sum` loop which iterates over a collection of objects and calculates their sum. For the sake of code reuse, the `sum` method can handle collections of any object type, as long as they implement the `add` method. In this example, the collection contains only `SmallInteger` objects. Since `SmallInteger` objects might be referenced from multiple places, the `add` method must create a new object, instead of modifying the `value` field in-place, which is a typical pattern in RPython VMS. Executing this loop without any further optimizations would create lots of intermediate `SmallInteger` objects that have no impact on the program state after the loop ends.

```
def sum(collection):
    if not collection:
        return SmallInteger(0)
    sum = collection[0]
    for i in range(1, len(collection)):
        sum = sum.add(collection[i])
    return sum

class SmallInteger:
    def __init__(self, value):
        self.value = value
    def add(self, other):
        return SmallInteger(self.value + other.value)
```

Listing 2.1: Simple `sum()` loop, using an aggregator variable

Listing 2.2 shows an exemplary trace of the `sum` loop. The trace is simplified by omitting the length-check of the collection at the beginning of `sum`; an actual trace would also contain more information in every instruction. The indented instructions are the contents of the loop. The `add` method has been inlined into the trace. After the `value` of the current element is extracted, it is added to the current `sum` value. Then a new `SmallInteger` object is allocated and the value is set. An escape analysis will determine that the temporary `SmallInteger` objects will not escape the loop, enabling the JIT to safely remove the red instructions and reduce the contents of the loop to a simple instruction sequence of `get_item`, `get_item` and `add`.

RPythons *meta* tracing JIT is a special case of a tracing JIT. Since RPython is used to implement VMS, it has to provide a JIT that is aware of the two levels of execution involved: the VM and the *user program*. A regular tracing JIT implemented completely inside a VM would detect loops and trace operations in the user program. A *meta* tracing JIT on the other hand detects loops in

```
...
sum = get_item(collection, 0)
label(loop, index = 1)
    i = get_item(collection, index)
    i_value = get_item(i, "value")
    sum_value = get_item(sum, "value")
    value = add(i_value, sum_value)
    sum = allocate(SmallInteger)
    set_item(sum, value)
conditional_jump_back(loop, index + 1)
return sum
```

Listing 2.2: Trace (in pseudo-instructions) of the loop in listing 2.1

the user program, but traces the operations performed on the VM level. This distinction is important for the following reasons.

Detecting loops on the user program level, instead of the VM level, delivers a better context for the JIT optimizations. The JIT requires as much contextualized information as possible, in order to achieve good optimization results. Such information is available in a program level loop because it can span over multiple bytecode executions, message sends and other VM level operations. The main loop on VM level, on the other hand, is often an infinite fetch-decode-execute loop. Since the *execute* part of this loop highly depends on the fetched bytecode, the execution path inside the loop will be different on each iteration, making it hard to optimize.

The tracing itself is performed on the VM level. Tracing operations on the user program level would introduce the semantics of the implemented programming language into the JIT. This would contradict the purpose of RPython to be a generic VM building toolkit. Also, more fine grained operations give more freedom to the optimizer.

Figure 2.4 shows the different levels of RPythons meta tracing JIT. The VM executes the bytecodes of the user program. The meta tracing JIT monitors the execution and waits for a back jump. In case of a bytecode VM, a back jump is detected when a jump leads to a smaller program counter than before. Once the JIT starts tracing the execution, the VMS operations are monitored and collected.

In order to detect loops in the user program, the JIT requires a minimum amount of information from the VM programmer. First, it needs to know which variables in the VM code identify the current execution state. This introduces a small piece of programming language semantics into the JIT. Typically, this would be an instance of the interpreter class, the currently executing method

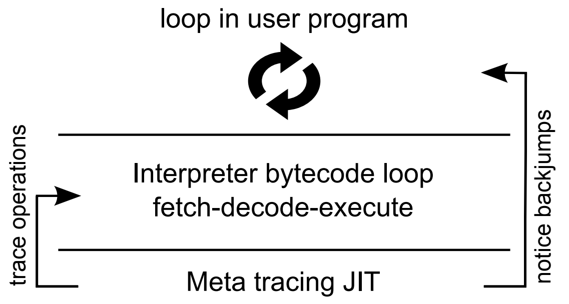

Figure 2.4: The levels in a VM with a meta tracing JIT
From top to bottom: the bytecodes of the user program, the VM executing the bytecodes and the JIT monitoring the two levels above.

object and a program counter pointing to a bytecode inside that method. The JIT also has to be notified when the user program makes a *back jump*. A back jump usually indicates a loop in the user program, and this is the moment where the JIT looks at the execution state and increments an internal loop counter. Finally, the JIT needs a second notification, usually right after a back jump, to let it know that it can start tracing the VM operations.

Although these hints introduce some JIT specific code into the VM, they are just a few method invocations. In return, the VM can be written in a much cleaner style, primarily expressing the semantics of the implemented language. For example, the VM can freely create intermediate objects without caring about performance: the JIT will remove unnecessary object allocations.

When creating machine code from a trace, the optimizer has to make assumptions about objects and variables in the trace. For example, the context of the loop might suggest that certain values do not change between iterations of the loop. The optimizer has to decide if a particular value is promoted to a constant, thus allowing for better optimizations, or if the value should stay a variable, thus making the current trace more general. Assumptions like this are enforced by inserting guards into the trace. Guards are assertions that fail if a certain condition does not hold. If a guard fails, the optimized trace is left and the VM falls back into the interpreted mode. The JIT keeps track of guard failures. When a guard fails too often, a new trace is created, inverting the assumption of the problematic guard. This way, an entire graph of traces is created after some time, covering all typical cases occurring at runtime.

2.2 Static Objects

The most important optimization implemented by RPythons tracing JIT is an aggressive escape analysis [14]. The escape analysis is conducted using a form of partial evaluation [3]. In short, it works by traversing the original trace, replacing object allocations with *static objects*, and turning a static object back to a normal allocation when it is observed to escape the trace. A static object is allocated directly on the stack, effectively avoiding the heap allocation. This optimization is safe because the optimizer can prove that static objects don't affect the program state after the program exits the loop.

For an example, consider the `SmallInteger` objects created inside the loop in listing 2.1 on page 9. These `SmallInteger` objects are not escaping the loop, because the only reference to each object is deleted at the end of each iteration. Only the object allocated in the very last iteration persists after the loop ends. Thus, all intermediate `SmallInteger` instances don't need to be allocated; they are treated as static objects. Their content is maintained on the stack instead of the heap, potentially even optimized into machine registers.

The drawback of static objects is a rather complex deoptimization. As mentioned above, an optimized JIT trace must be left when a guard check, inserted to ensure correctness of the code, fails. In such a case the escape analysis performed as basis for the static object optimization does not hold anymore. Static objects allocated in the last loop iteration have to be allocated on the heap, and their contents have to be copied from the stack to the newly allocated objects. To do this, the JIT has to maintain information about the stack layout while executing an optimized trace; using a stack map, the stack can be walked in order to create non-static versions of the objects.

Since RPythons meta tracing JIT creates traces of VM operations, not only language level objects can be turned into static objects, but also VM level objects. Just like other optimizations in the RPython toolchain, this leads to cleaner code; short living objects can substantially increase the expressiveness of VM code.

2.3 Allocation Removal on VM level

The previous section described some allocation removal optimizations performed automatically by RPythons JIT compiler. However, allocation removal has a lot of potential that is not covered automatically by the RPython toolchain. One important reason is that there should be a limit to the JITS complexity. Additional optimizations require additional analyses which have to pay off when executing the optimized machine code. The more complex and long running an optimization or analysis gets, the less likely it is to be worth while. Optimizations on a general level like the JIT should not be too specific for the same reasons; an analysis that rarely produces any results will just waste computation time during the warmup phase.

Language specific optimizations should be implemented on the same level that also defines the language semantics. Therefore, to further exploit allocation removal techniques, some optimizations have to be implemented on the VM level. This works well since the VM already handles all allocations performed by the user program, and object allocation is one of the most frequent operations. The VM also has detailed knowledge about the memory representation in the user programming language. Therefore, the VM can transparently replace physical allocations with an alternative operation providing the same effect. This section describes some allocation removal techniques that are best implemented on VM level.

2.3.1 Pooling Immutable Objects

The *pooling* allocation removal technique is based on the *flyweight* software design pattern described by Gamma et al. [13], also known as the Gang of Four. A flyweight object must be safely reusable in different contexts simultaneously. Therefore a flyweight object must have some context-independent, *intrinsic* state which is typically immutable. All additional *extrinsic* state must be maintained outside the object and passed along with method invocations. Flyweight objects can be obtained using a factory object instead of creating new instances every time an object is required. The factory has two important tasks. First, it has to map creation parameters to actual flyweight object instances. Typically, the creation parameters identifying an object are the constructor parameters. These values also define the intrinsic state of a flyweight object. Second, the factory has to actually create the flyweight instances. This can mainly happen based on two strategies: either preallocate all possible instances or create instances lazily, on-demand. The first approach can only be implemented when the value ranges of all creation parameters are known, finite and not unreasonably large. In return, accessing a flyweight object is very fast because all objects are created in advance. The second approach is the more general one, but should be avoided in very performance critical situations.

In order to apply the flyweight pattern on VM level, two strict conditions must be met. The pooled objects must be immutable, and the *identity* of the objects must be fully defined by their *value*. The first condition is directly based on the flyweight pattern: if the same mutable object is used simultaneously in more than one context, there will be unexpected consequences. The second condition is more VM specific: object identity is a kind of unique object state which is required by most object-oriented (and even non object-oriented) programming languages. There are few exceptions to this, which are mainly numeric primitive datatypes like bytes, characters, booleans or numbers. In Java, for example, primitive types like `int` don't even have anything like object identity, and in Smalltalk instances of `SmallInteger` are always identified by their value.

To implement a pooled datatype, the VM must play the role of the factory in the flyweight pattern. The VM programmer has to identify all places where objects of that type are created, are replace the object allocation with pool accesses. Since this is a performance optimization, a very efficient pool access should be implemented. Therefore, the mapping from creation parameters to object instances should be a very simple operation. The quickest way to do so is to convert the parameters to an array index and store the instance at that index, thereby eliminating the need of a full-featured hash table. Also, the instances should be preallocated, if possible. Luckily, the mentioned primitive datatypes fulfill these conditions perfectly: the value of a character or integer, which is the only creation parameter, can be directly used as array index. In the case of characters or bytes, the value range is small enough to preallocate every possible instance. Integers, on the other hand, have a larger value range, so that only a certain range, like the lower 1024 integers, can be pooled that way.

Many languages also have immutable string datatypes, clearing the first condition for object pooling. The second condition of defining object identity based on object value is also given in some cases. Javas `String` literals, for example, are automatically interned and equal `String` values share the same object. In this case the VM *has to* implement an object pool for `String` objects. Therefore, this case is an example of implementing correct language semantics, instead of an actual performance optimization. Preallocating additional string objects based on heuristics, similar to integer objects, does not seem feasible.

2.3.2 Pointer tagging

One of the oldest allocation removal techniques is pointer tagging [16]. Pointer tagging is exploiting the fact that machine addresses are typically word aligned. This means that the two least significant bits of pointers on a 32 bit machine are always zero. On a 64 bit machine the last 3 bits always remain zero. These bits can be used to form a tag on every single pointer in a running program. That tag can for example indicate the type of the pointer. In an object-oriented language, one type should indicate a regular object reference, but other tagged types can be numerical datatypes or booleans. The only condition is that values of the tagged datatypes fit into a reduced number of bits, since part of the machine word is used for the tag.

The most common form of pointer tagging is integer tagging. Since integer arithmetics is an important part many programs, VMS often put a high priority on its performance. In many dynamic object-oriented languages, even primitive number datatypes are semantically objects, and arithmetic operations on these types are regular message sends. Polymorphism and the overhead of message sends makes these operations slow. To optimize integer operations, the integer can be stored directly in the pointer word, using pointer tagging. This reduces the number of allocations, since new integer objects are created simply by tagging pointers. Also, arithmetical operations are faster because of the missing indirection between the pointer word and the object data.

In theory, pointer tagging can be used to identify up to 4 different special in-place datatypes on 32 bit machines, and up to 16 datatypes on 64 bit machines. Potential candidates include boolean objects, byte or character objects and enumeration datatypes that can take on a fixed number of values.

While this may sound promising, there are also drawbacks to pointer tagging.

PRECISION. The precision of integers (or other tagged datatypes) is reduced to 31 bits, or even less. Many programmers and programs assume integer precision to be based on the size of machine words, so the VM will have to handle the case of overflowing integers correctly.

PERFORMANCE. Integer operations might be very common, but accessing object data via an object reference is also a frequent operation. The VM must test for tag bits on every single pointer dereferencing. If no integer objects are encountered in a long time, this imposes pure overhead on one of the most common operations in the VM. In order to pay off, pointer tagging (or integer tagging in particular) has to be used in situations where integer operations happen often enough.

COMPLEXITY. Pointer tagging introduces complexity into the core of the VM. Since object pointers are handled at many places in the VM, the entire code base gets littered with case differentiations regarding the handling of such pointers. This is a cross-cutting concern and hard to express in a single place in the code base.

While the RPython toolchain has a mechanism to implement a single tagged datatype, it has not been used for the PyPy interpreter. Bolz et al. showed that RPythons tracing JIT in combination with another optimization, storage strategies, makes tagged pointers unnecessary [5].

2.3.3 Storage Strategies

Usually, collections in dynamic languages are represented by allocating an array and filling it with pointers that point to the actual elements of the collection. This is because in a dynamic language, or even any language with polymorphism, objects of different types can be added to the same collection. Using an array of pointers is the generic way to represent such a dynamic collection. However, it also introduces an indirection between the collection and the actual elements. To access any value from the collection, the VM must first fetch the pointer address and dereference it, before fetching the actual value. This additional memory access is a big overhead compared to a C array of pure integer values, that a program can traverse in full speed.

Another allocation removal technique called **storage strategies**, or **strategies** for short, can optimize dynamic collections [5]. Storage strategies are implemented on VM level and are based on the idea that the VM has multiple possible ways of representing every collection internally. Based on the user program and the operations performed on a collection, the VM can switch between multiple strategies and choose the most efficient one. To do so, the collection contains a reference to the currently used strategy object, and all important operations performed on the collection are delegated to that object. Ideally the strategy will not be switched very often, so the second important idea behind storage strategies is that user programs often utilize collections in a predictable way.

For example, collections are often used in a homogeneous way. If the first few elements appended to a Python list are integers, then it is likely that the user program will keep storing only integer values in that list. This is enough information to optimize the list. In this case, the VM could allocate an array for integer values and store the values directly, without allocating box objects to represent these integers. Figure 2.5 illustrates this example by showing two possible representations for the same collection: the collection on the left side contains boxed integer objects, while the collection on the right side contains pure integer values.

In order to optimize the internal representation of a collection, the VM must make assumptions about the usage pattern of that collection. It cannot wait until a collection has been used for a while before optimizing it, otherwise it might end up not optimizing short living or small collections. Optimizing short living collections can be very important, for example when they are allocated frequently in a tight loop. In order to optimize as many collections as possible, the VM must predict the usage pattern based on the very first operations performed on a collection.

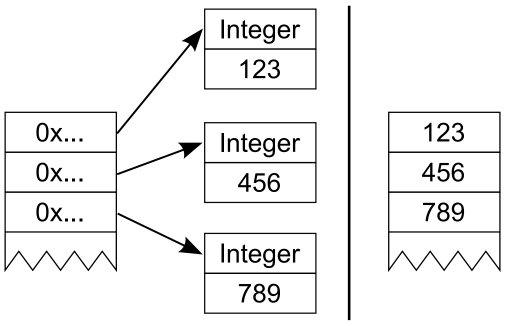

Figure 2.5: Example for a storage strategy
Left side: generic collection holding only boxed integer objects.
Right size: Optimized collection, resulting in less object allocations and memory usage.

Making predictions about usage patterns is hard and the success can vary depending on the situation. Therefore, the VM should make assumptions based on meaningful heuristics, which can be obtained by examining existing systems and deriving typical usage patterns. But even with these preconditions there will be cases where the VMS assumptions turn out to be wrong. In such cases, the VM must uphold the semantics of the programming language and perform a *deoptimization*. Most of the time this means allocating a new, more general collection and copying all elements from the old collection into the new one. Figure 2.6 shows an example of this procedure. The storage on the left is optimized to hold integer values, but the user program wants to add a float value to the list. The VM has to allocate a new, more general storage array which is able to hold generic objects. In this case, the deoptimization could have been avoided if the VM had implemented a storage strategy to hold both integer and float values. The deoptimization procedure must be tuned for highest performance, and should take place as rarely as possible.

Homogeneous Collections

As indicated by the examples in the previous section, the most important optimization target of storage strategies are homogeneous collections. A collection is homogeneous when it only contains elements of a single type. Since the size of the datatype is known, the VM can allocate an array big enough to hold the data of many such objects in consecutive memory. The benefit of this optimization is two-fold: the VM can save the memory for the objects, and accessing the objects becomes faster due to the missing indirection between collection and elements.

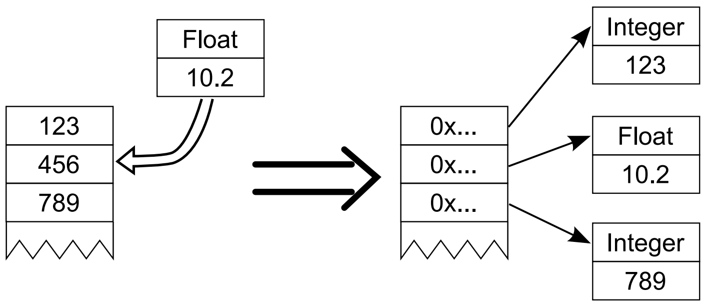

Figure 2.6: Deoptimization of an optimized integer collection
A general storage array has to be created, and Integer objects have to be allocated.

This optimization is not limited to numeric datatypes like integers. Even objects with multiple fields can be represented that way, as illustrated in figure 2.7. The fields of each objects just have to be placed in memory in sequence. Accessing any field in any object can happen without additional memory access, simply based on constant-time pointer arithmetics.

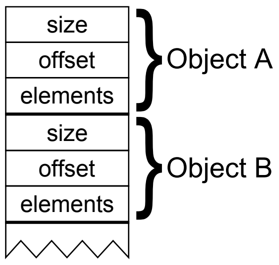

Figure 2.7: Inlining storage for composite objects with 3 fields

3 Generic Storage Strategies

Programmers expect modern VMS to execute their code with good performance regarding memory and computational time. The performance of a program often imposes an important secondary requirement, next to the programs primary functionality. In the context of real-time systems the performance is even as important as the main functionality. Achieving performance requirements is a challenging task and often regarded as the most difficult part challenge in software engineering. Therefore, programmers often rely on the VM to provide a high level of efficiency.

The main obstacle that the VM must overcome is the difference between the programming language and the underlying hardware. The VM has to map the two abstractions in order to execute a program. Many high-level operations of programming languages can not be mapped directly to machine code operations. Instead, the VM has to *emulate* programming language semantics on top of the physical machine by executing entire subroutines even for the most frequent operations in the program.

Providing good performance becomes even harder when programmers are not aware, or do not want to be aware, of the underlying hardware. Dynamic languages include features for writing abstract, reusable code and writing code in this style is considered good practice. However, the high-level operations in such dynamic languages are even farther away from the underlying execution platform.

Collections are an important part of dynamic programs and especially sensitive to poor performance. Working with collections often means executing routines on every single member of the collection, which multiplies the effect of an inefficient routine.

In order to achieve good performance, many languages and standard libraries provide rich collection libraries that allow efficient representation of program data, are designed to work well on modern hardware or simply provide well-designed algorithms. However, this is not an automatic process: the programmer has to consciously select the collection or algorithm best suited to solve the problem at hand. Such design decisions can clutter the code and make it less reusable and abstract. The programmer can also simply be unaware of the multitude of possibilities and their potential benefits. Therefore programmers often use simple, unoptimized collections. The VM should still try to optimize the inefficient code internally; the penalty for writing clean code should be kept as low as possible. If a programmer does choose to optimize the code, it should of course still benefit of these optimizations.

In this work we describe storage strategies as an optimization that can help the VM to achieve the expected level of performance. In short, the VM chooses optimized internal representations for certain datatypes based on heuristics, while still maintaining the semantics of the programming language. This optimization helps the VM solve the two aforementioned problems in the following ways.

THE ABSTRACTION GAP The VMS assumptions when implementing storage strategies allow it to trade certain dynamic properties for a more optimized internal representation which is closer to the underlying hardware.

INEFFICIENT CODE The VM can detect usage patterns for certain data structures and automatically apply optimizations that could otherwise be applied on user program level. Since the VM has less information available than the program itself, it has to use heuristics and make predictions.

In order to aid the construction of high performance VMS, The RPython toolchain implements many automatic optimizations at compile time and runtime. When translating RPython to C, the code is optimized based on static analysis. Later, at runtime, the tracing JIT compiler is able to perform even more aggressive optimizations based on runtime type and value feedback.

Despite their potential, storage strategies are not automatically provided by the RPython toolchain. Neither is there an implementation in the library for RPython based VMS, called rlib.

Different programming languages have different datatypes, collections and requirements. Still, every programming language has some notion of collections. We show that it is possible to identify and extract reusable aspects of collections and combine them into a library for storage strategies. Such a library can be independent of programming languages and be reused by different RPython based VMS.

3.1 Library Design

In this chapter we present the design of a library for storage strategies called rstrategies. First we introduce the principles this library design is based on, then we describe different parts of the library and how a VM developer can utilize the library to create specific collection implementations.

In order to be reusable in the context of different programming languages, rstrategies must expose its functionality in a very flexible way. We use the concept of mixins [7, 11] to provide the required level of flexibility while allowing the VM to extend the provided functionality when needed.

In the context of RPython, any regular class can be used as a mixin. The programmer must simply include a `import_from_mixin(MixinClass)` clause into the body of the instantiating class. All attributes and methods will then be copied over from `MixinClass` to the target class, including all inherited methods. A mixin is not intended to be used on its own. It can require other mixins and fields or methods provided by the instantiating class itself. This way the code inside a mixin can be written in an abstract way to promote modularity.

rstrategies supports two types of collections: fixed sized and variable sized lists. A fixed sized list is a linear one-dimensional collection with a fixed size that cannot be changed after creating the list. A variable sized list extends this concept by allowing to dynamically change the collection size by adding or removing elements. These two abstract types of collections cover the needs of primitive collections for most programming languages. The reasons behind this design decision are further explained in section 3.4.

The rstrategies library provides a hierarchy of mixins that can be combined into actual collection classes. If required, parts of the provided functionality can instead be implemented in the VM specific collection classes. Any mixin class can also be extended in the VM specific code before being used. At the top of the mixin hierarchy is the class `AbstractStrategy`.

`AbstractStrategy` defines the interface implemented in its specific subclasses, in addition to a few internal methods. VM code will mainly interact with rstrategies and the various storage strategy implementations through the public interface defined in `AbstractStrategy`. The following list summarizes the five methods defined in `AbstractStrategy`:

STORE(INDEX, ITEM) Stores the given `item` at the given `index` in the collection. The strategy will check internally if it can store the value of `item` in an optimized way. If this check fails, the strategy will be automatically switched to a more general strategy that is able to hold both the current elements and the new `item`. The switching mechanism is explained in section 3.3. Optionally, the value of `index` is checked and an error is raised if it is out of bounds of the collections size. The value of `index` must be within the bounds of the collection, even if the collection is variable sized.

FETCH(INDEX) Returns the item stored at the given `index` in the collection. Depending on the underlying strategy, it might be necessary to box the value on the fly to represent it on the VM level. Because of this, the results of

EmptyStrategy	Size always zero
SingleValueStrategy	Every element is the same value
GenericStrategy	Can hold any number of different values
WeakGenericStrategy	Allows its elements to be garbage collected
SingleTypeStrategy	Stores unboxed values of a single type
TaggingStrategy	Stores unboxed values of a single type plus one special tag value

Table 3.1: Summary of the strategies supported by rstrategies

two identical calls to `fetch` might have different object identities, while still always containing equal values. The distinction between object identity and value equality is important. The same goes for a call to `fetch` after switching the underlying storage strategy: the results must contain equal values, but there are no guarantees about the object identity.

SIZE() Returns the number of elements stored in the collection. In case of a variable sized collection, the result of this can change after an intermediate call to `insert` or `delete`.

INSERT(INDEX, ITEMS) Inserts the given list of `items` at the given `index` in the collection. This increases the size of the collection by the number of inserted items and should therefore only be used in the context of variable sized collections.

DELETE(START, END) Deletes the items at indices `start` (inclusive) to `end` (exclusive) from the collection. All indices must be within the bounds of the collection. This decreases the size of the collection by `end-start`.

3.2 Strategy Optimizations

The interface of `AbstractStrategy` is designed in a way that any of its subclasses can be used both in a fixed sized or variable sized context. A collection becomes fixed sized by simply never invoking any of the `insert` or `delete` methods. RPythons optimizer will then determine that underlying storage arrays are never changed in size and accordingly optimize allocations and accesses of such collections.

Different subclasses of `AbstractStrategy` represent different storage strategies, as described in chapter 2.3.3. Table 3.1 gives an overview over the supported strategies, which are further explained in the following paragraphs.

The first strategy is `EmptyStrategy`, which represents a collection with zero elements. The size of a collection using `EmptyStrategy` is always zero, the `store`, `fetch` and `delete` operations always raise an error, and the `insert` operation always switches the strategy to a different one. Since these methods do not require any state, the `EmptyStrategy` does not allocate any storage and can save a lot of memory when used frequently. The `EmptyStrategy` can be used together with variable sized lists, like Pythons primitive `list` type. In such a context, the `EmptyStrategy` would be the starting point for every empty list. Then, depending on the first inserted element, the storage strategy would be switched appropriately. When all elements of a list are removed, the strategy can be optimized to the `EmptyStrategy` once again, in case it is reused with a different type of element.

Another useful strategy mixin is `SingleValueStrategy`. This strategy can have an arbitrary size, but can only store the same value one every slot in the collection. Therefore it does not require to allocate an entire storage array. Again, this strategy can save a lot of memory when used frequently. The `SingleValueStrategy` must allocate memory only for a single value: the size of the list. The single value stored in this strategy is a compile-time constant. As soon as a different value is stored, the collection must be deoptimized to a more general strategy. This is useful for programming languages with fixed sized primitive collections that are always instantiated filled with a default element. For example, a Java `Array` is initially always filled with `null` values, and a Smalltalk object is initially always filled with `nil` values. Here, the `SingleValueStrategy` can be used as initial strategy, before being deoptimized to a more general strategy.

`GenericStrategy` is the strategy that is able to handle any number of different values. It corresponds to a simple, non-optimized list and is the fallback strategy when any other strategy has to be deoptimized. Depending on the context, there can be cases where a `GenericStrategy` is again optimized to another strategy. For example, if the last element is removed from a variable sized list using a `GenericStrategy`, it switches the strategy to `EmptyStrategy`. This way, it will again switch to another optimized strategy next time an element is added. Similar scenarios can be constructed when using a fixed sized list, but in that case additional information must be maintained about the elements. For example, keeping track of the number of `null` values would allow optimizing a generic fixed-sized array to `SingleValueStrategy`. The resulting overhead of maintaining this information would most likely surpass the optimization benefits.

`WeakGenericStrategy` is a version of `GenericStrategy` that holds on weakly to its elements. This means that an element of a list using a `WeakGenericStrategy` can be garbage collected, if the list contains the last reference to this element. Many programming languages require support for weak collections or objects on VM level. Since this strategy serves a very special purpose, collections should never switch from or to this strategy. Instead, a collection should be instantiated with this strategy and keep it throughout its lifetime.

`SingleTypeStrategy` is able to store elements of a single type in an optimized storage array, as described in section 2.3.3. Certain conditions must be met in order to use this strategy. Namely, the identity of the element objects must be determined only through their value, and element objects must be immutable. These strict conditions are the same as for pooling objects (see section 2.3.1) and are usually only met by primitive types like integers or floats. Therefore, the `SingleTypeStrategy` can be used to implement optimized storage for collections of such primitive datatypes. The VM has to provide routines for *boxing* and *unboxing* the elements of this strategy. These routines form the bridge between the machine level and VM level representations of the element values. For example, assume an `IntegerStrategy` has been implemented based on `SingleTypeStrategy`. Now a boxed integer object is added to a collection with such a strategy. This object has to be unboxed in order to store its value into a machine level array of integers. When fetching a value from this strategy, the according boxed integer object has to be created again.

The `TaggingStrategy` is an extension of `SingleTypeStrategy`. In addition to optimized storage of a single element type, it uses one specific tag value in the value range of unboxed elements to represent some special boxed value. This is useful when fixed sized collections have a default value for their contents. For example, a fresh Java `Array` is filled with the default value `null`. When this array is filled with `Integer` values, there will often be some slots in the array which still have the value `null`. Now if we were using the `SingleTypeStrategy`, we would already have to deoptimize to `GenericStrategy` in order to represent both `null` and `Integer` values in the same collection. With the help of the `TaggingStrategy`, we can avoid this early deoptimization by representing the `null` slots with special integer values, for example the maximum possible integer value.

There are two drawbacks to this strategy. First, it adds the overhead of comparing fetched elements with the tag value. This is the penalty for representing two different datatypes in the same optimized storage. Second, the `TaggingStrategy` can not store the tag value itself. Therefore, the tag value should be chosen to occur very rarely in real situations. This can be hard for integer values, since even the maximum possible integer can very well be produced by the user program. For float values, the IEEE 754 Standard [17] allows for multiple representations of the `NaN` value, one of which can be used as tag value.

The described strategies can further be combined by the VM. For example, a Smalltalk object can consist of two parts: a fixed sized part containing instance variables, and an optional variable sized part, which is Smalltalks notion of arrays. Even though the two parts form a single object, they can have different requirements regarding storage strategies. In this case, the VM can use two instances of different strategies to represent a single object.

The VM can even implement strategies that contain objects with more than one data field. For example, a language might have a primitive type representing a *point*, consisting of two number fields. A collection of such *point* objects could be optimized by storing all the point numbers in one consecutive array. This can be implemented using `SingleTypeStrategy` and storing instances of RPythons `tuple` type into it. In this example, the VM would have to provide routines to convert between the tuple of raw data, and an actual *point* object.

3.3 Switching Strategies

An important part of rstrategies is the mechanism that allows a collection to switch between different storage strategies. The generic and naive way to switch from one strategy to another requires two steps:

1. Allocate a new storage strategy of an appropriate type and with an appropriate size.
2. Invoke `fetch` on the old strategy instance and `store` on the new strategy instance for every element in the old instance.

While the first step can not be avoided, the second step can be largely optimized in many cases. For example, when switching from a `SingleValueStrategy` to a `GenericStrategy`, every `fetch` operation will yield the same result for the old strategy instance. Therefore, an optimized switching routine can be implemented, which allocates a `GenericStrategy` and fills its storage array with a fixed value, using an efficient routine similar to `memset` of the C standard library.

In rstrategies, such optimized switching routines are implemented using a `StrategyFactory`. An instance of `StrategyFactory` knows all available storage strategies, manages the switching of strategies and instantiates new strategies. The `StrategyFactory` class is described in more detail in section 4.2.1.

3.4 Discussion

This section summarizes some design decisions and intentional limitations which we built into the rstrategies library design. We explain our motivations for the various decisions and discuss alternative approaches. The points discussed here mainly concern the kinds of collections and strategies we chose to support.

3.4.1 Supported Collection Types

The collection types supported by rstrategies are fixed sized and variable sized lists. Of course there are may more collection types and algorithms used in modern programs. The collection framework in the Java standard library features 8 interfaces and 10 implementation classes. A modern Squeak image[1] has as many as 79 subclasses of `Collection`, because it includes collections with very special purposes. However, rstrategies is a library for optimizing collections on VM level, which means it is only meaningful to support collections that occur as *primitive types* in multiple programming languages. Even though Java supports collections like `ArrayList`, `TreeSet` and `HashMap`, the only primitive collection type is the `Array`. The same goes for Smalltalk and many other programming languages.

It would be possible to take the approach of storage strategies even further and let the VM transparently optimize collection types from the standard library of the programming language. For example, since the interface and semantics of Javas `ArrayList` are well defined in principle, the VM could try to handle instances of `ArrayList` by itself, without actually executing the Java bytecode in the methods of the `ArrayList` class. This might lead to improved performance when accessing elements of `ArrayList`, but this approach has several drawbacks. One important function of a standard library is to reduce the complexity of the VM. Moving parts of the library into the VM would only further increase its complexity. Another purpose of having a separate standard library is that it is able to evolve independently of the VM. It is very hard to exactly mirror the code of the standard library and the suggested approach would require releasing a new VM version after every substantial change to the standard library.

Therefore we chose to only optimize primitive collection types with rstrategies.

3.4.2 Non-List Collections

In the following we further motivate our decision to only support fixed sized and variable sized lists in rstrategies.

We examined the primitive datatypes of the top programming languages according to the TIOBE Index for July 2014[2]. Even though such an index is

[1] `http://www.squeak.org/` (August 2014)

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (August 2014)

Language	Fixed List	Variable List	Dictionary
Java	Object, Array		
Objective-C	Object, Array		
C#	struct, Object, Array		
PHP		array	array
Python	tuple, bytearray	list	object, dict
JavaScript		object	object
Perl	Object, array		hash
Visual Basic .NET	Object, Array		
F#	list, tuple, array		struct, record
Ruby		array	object, hash

Table 3.2: Programming languages and their primitive collection types

never exact and changes fairly quickly, we believe it provides a satisfactory view of the state-of-the-art of programming languages. We analyzed the top 10 programming languages which are usually executed by a VM. C and C++ are excluded, for example, since storage strategies are not applicable for statically compiled languages: these languages include explicit semantics of memory layouts, making it impossible to dynamically change them. Table 3.2 summarizes the results by listing the primitive datatypes of each language and mapping them to abstract collection types which are explained in the following.

The first line of table 3.2 lists 3 abstract collection types: *Fixed List*, *Variable List* and *Dictionary*. *Fixed List* and *Variable List* correspond to the collection types supported by rstrategies, using the mixin hierarchy described in 3.2. Therefore, all collection types listed in these two columns could be implemented and optimized using rstrategies.

Dictionary is a third abstract collection type that is quite important for many languages. There are two kinds of *Dictionary* based collections. The first kind corresponds to Pythons or Rubys `object` type, which only allows symbols as keys. This type is an integral part of these languages and therefore its optimization is central to VM performance. The standard optimization for such *slot objects* is called maps, has been first developed for the Self VM [9, 8]. This advanced optimization is not part of rstrategies because it is inherently language dependent. The second kind of *Dictionary* based collections are typical lookup tables as implemented in standard libraries of many languages. This collection type is supported as built in type in Python, Perl and Ruby. Other languages, like JavaScript, feature types which can be used as both slot objects or general lookup tables.

Storage strategies can be extended to include support for lookup tables. This has been done for the Python interpreter PyPy [5] and in the JRuby+Truffle project [22]. We have not yet implemented lookup tables as part of rstrategies, but intend to do so as part of our future work on this library. We expect this extension to prove especially useful for languages like Ruby, where small `hash` values are often used to pass parameters into a method.

The contents of table 3.2 are complete for every language, with one exception. Python has several additional primitive types for special use cases: `buffer`, `xrange`, `set`, `frozenset`, `iterator`, `generator`. This variety of primitive collection types is special to Python and all these types can be mapped to the *Fixed List* or *Variable List* types by adding additional algorithms to the `store` and `fetch` routines. Therefore we chose not to explicitly support special datatypes like these.

3.4.3 Multilevel-Strategies

In theory, storage strategies could even be applied to multiple levels, like illustrated in figure 3.1. In this example, there are multiple equally sized lists of integer objects, which in turn are collected in a single, larger list. This two-level setup could be optimized by placing the unboxed version of all integers in consecutive memory.

We decided to not take the storage strategies approach that far. For one thing, scenarios like that are rare. Furthermore, this optimization would require the VM to store and constantly evaluate additional control information, like the number of levels optimized and the length of the contained lists. This requires additional resources and the required analysis would likely be too expensive to pay off.

3.4.4 Non-Contiguous Lists

Functional programming languages often feature a primitive datatype for linked lists. Lisp for example has the built in type `cons` which represents one link of a linked list, formed by two fields: the value (named `car`) and the pointer to the next link (named `cdr`). While these labels have historical reasons originating in the 1950s, the linked list is still a central data structure in modern Lisp programs. Since accessing arbitrary elements of a linked list requires walking the entire list up to that element, optimizing linked lists would be very beneficial. For example, all elements of the list could be placed in consecutive memory, allowing constant access to any element. However, such optimizations bring about further complications and are a separate (although related) research

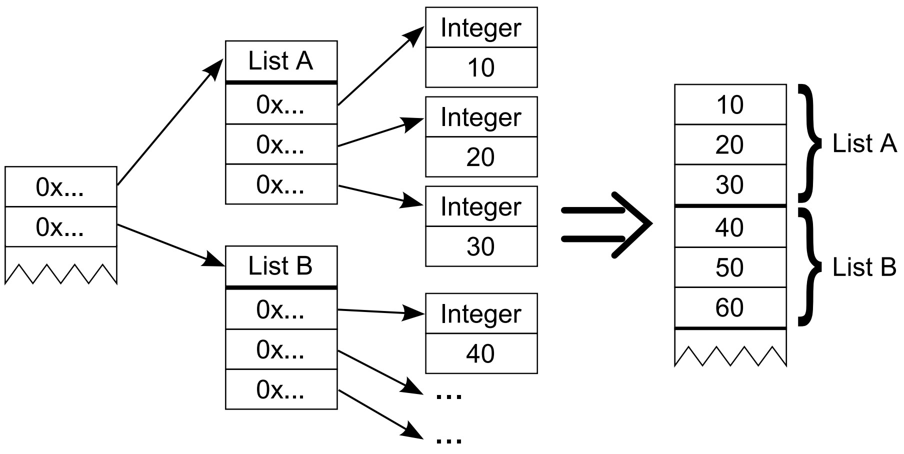

Figure 3.1: Theoretical optimization: multi-level storage strategies

topic to storage strategies. Therefore, we consider optimizations of linked lists out of scope for rstrategies.

4 Implementation

This chapter describes the implementation of the **rstrategies** library that has been written in RPython. **rstrategies** has been written to be part of the **rlib** library, which is the general support library for VMS written in RPython. Section 4.1 describes **rlib** and some of its main components, section 4.2 gives implementation details about **rstrategies** itself, and section 4.3 shows how **rstrategies** has been adopted in several existing RPython VMS.

4.1 Rlib

rlib is the main library used by RPython VMS and is currently hosted in PyPy's main repository[1]. **rlib** is under active development and the descriptions given here can only represent **rlib** at the time of this writing. This section describes the purpose of having a dedicated library and the kind of code and application programming interfaces (APIS) the library contains.

One of the purposes of **rlib** is to collect RPython code that might be useful for multiple RPython VMS.

Since the RPython programming language is a restricted subset of Python, not every existing Python library can automatically be used in an RPython program. Most libraries take advantage of Pythons very dynamic features to present the user with a flexible and comfortable API. These dynamic features are not part of RPython, making many libraries unusable in the context of RPython VMS.

When VM developers discover a library useful for VM development, they usually follow one of two common procedures to add it to **rlib**. The first way is to copy the library code with the permission of the original authors, modify and refactor it until it is completely valid RPython, then check it into the **rlib** repository. This way is only feasible if copying the foreign code is appropriate, and if the library is small enough to manually translate it to valid RPython code. The second way is to write an RPython version of a certain library from scratch. In practice, a mix of the two procedures is mostly used.

[1] `https://bitbucket.org/pypy/pypy` (August 2014)

The second important purpose of the rlib library is to provide APIS to interact with the JIT and other parts of the RPython toolchain. These interfaces are important when it comes to optimizing the VM for performance, but can also be useful in order to create well-formed RPython code.

The following is a list of some of rlib's modules and short explanations of their purposes.

PARSING A toolkit for creating simple parsers for AST construction. Most VMS use some sort of parser in their front-end, the notable exception being pure bytecode VMS like RSqueak.

JIT This module is the main interface to interact with RPythons tracing JIT. The `JitDriver` class must be used in order to provide the JIT with some mandatory information. Other methods can be used to tweak the performance of certain hot spots: `purefunction` marks the return value of a function to be constant given a fixed set of parameters, `unroll_safe` tells the JIT that loops in a method can be safely unrolled and `dont_look_inside` prevents the JIT from tracing inside a method. There are also query methods like `we_are_jitted`, which tells the VM if it is currently executing optimized machine code.

OBJECTMODEL This module contains APIS to interact with the translation process and underlying runtime of the translated VM. `specialize` can be used to create multiple versions of a method, based on certain conditions like the types of the parameters; it is an important tool to create statically typed RPython code. `we_are_translated` returns a boolean value indicating whether the VM is currently running in translated mode, or on top of a Python interpreter. `compute_identity_hash` can be used to obtain a unique hash value for a given object. `import_from_mixin` is the main method for implementing mixins in RPython - a mechanism that is heavily used in the implementation of rstrategies.

RTHREAD, RFILE, RSOCKET, RPATH, RPOSIX These and other modules provide RPython-conform interfaces to the underlying platform.

4.2 Rstrategies

Chapter 3 gave an abstract description of the architecture of rstrategies and all of its components. In this section we present some of the elements of rstrategies in more detail combined with source code excerpts.

The main functionality of rstrategies is provided in form of one class hierarchy shown in figure 4.1.

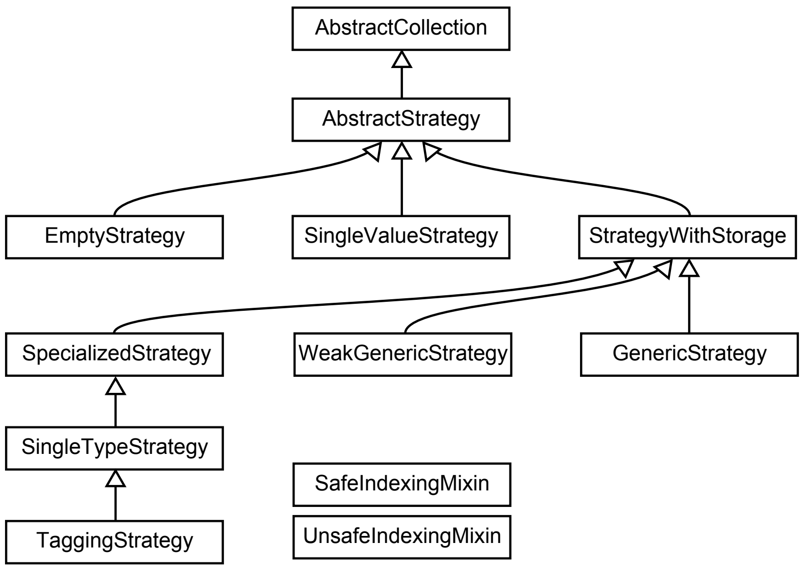

Figure 4.1: The main classes of rstrategies

Abstract classes in this hierarchy always define the abstract methods that are implemented by a concrete subclass. Also, since these classes are used as mixins, they always contain comments which describe methods that are required for the particular class to work. For example, listing 4.1 shows a part of the definition of `AbstractStrategy`. The methods `store`, `fetch`, `size`, `insert` and `delete` form the main interface provided by the non-abstract strategy mixins. `check_can_handle` is a boolean method that returns `True` if the strategy is able to store the wrapped object passed as `value`. It is an important part of the internal mechanism of selecting and switching strategies. In most cases it is simply implemented by a type check using Pythons built-in function `isinstance`. The comment in listing 4.1 shows that `AbstractStrategy` requires two methods in order to function: `strategy_factory` and `__init__`. These two methods must be provided by any class using this mixin. Since they will likely be the same for every collection class in a given VM, the collection class hierarchy should have a single class at the top. This root class should include the `AbstractStrategy` mixin and provide basic methods like `strategy_factory`. The root class of the collection hierarchy is also important for the `StrategyFactory` which is described in the next section.

The simplest strategy mixin is the `EmptyStrategy`, shown in listing 4.2. It cannot handle addition of any values and always has the constant size of zero. This is also what makes this strategy very efficient: no storage is allocated at

```
class AbstractStrategy(object):
    """
    == Required:
    strategy_factory(self) - Access to StrategyFactory
    __init__(...) - Should invoke init_strategy(self, size)
    """

    def store(self, index0, value):
        raise NotImplementedError("Abstract store")
    def fetch(self, index0):
        raise NotImplementedError("Abstract fetch")
    def size(self):
        raise NotImplementedError("Abstract size")
    def check_can_handle(self, value):
        raise NotImplementedError("Abstract check_can_handle")
```

Listing 4.1: Part of the definition of `AbstractStrategy`

```
class EmptyStrategy(AbstractStrategy):
    """
    == Required:
    See AbstractStrategy
    """

    def fetch(self, index0):
        raise IndexError
    def store(self, index0, value):
        self.cannot_handle_value(index0, value)
    def size(self):
        return 0
    def check_can_handle(self, value):
        return False
```

Listing 4.2: The code for `EmptyStrategy`

all and most operations on it return constants. This makes it a good starting point for collections that have a chance of staying empty.

Other strategy mixin classes are more complex. The most useful implementations of `fetch` and `store` are part of `StrategyWithStorage`. They are shown in listing 4.3. First they both execute an index check which raises an Exception in case of an invalid invocation. The `store` method then checks if the current strategy can handle the incoming value. If so, it unwraps the value and stores it in the storage array. Otherwise, the `cannot_handle_value` routine is called, which ultimately makes the collection switch to a more general strategy. This piece of code is very powerful since it describes the main concept of storage strategies for all types of contained values. This is normally not possible in RPython because of its static type checker. This code would be invalid as soon as there are multiple inconsistent usages of the `self.storage` field. However, since this code is used in a mixin scenario, it is copied to multiple places before the type checking is performed. This way we can write weakly typed code like this even in this statically typed language. The `fetch` method in listing 4.3 fetches an element from the `self.storage` array and returns it as a wrapped object. `StrategyWithStorage` requires `_wrap` and `_unwrap` routines that are able to convert between the stored value and a wrapped representation.

```
def store(self, index0, wrapped_value):
    self.check_index_store(index0)
    if self.check_can_handle(wrapped_value):
        unwrapped = self._unwrap(wrapped_value)
        self.storage[index0] = unwrapped
    else:
        self.cannot_handle_value(index0, wrapped_value)

def fetch(self, index0):
    self.check_index_fetch(index0)
    unwrapped = self.storage[index0]
    return self._wrap(unwrapped)
```

Listing 4.3: `fetch` and `store` from the `StrategyWithStorage` mixin class

A complex example of a `check_can_handle` method can be found in the `TaggingStrategy` mixin, as given in listing 4.4. The `value` object must fulfill three requirements in order to guarantee correct semantics of this strategy.

```
class TaggingStrategy(SingleTypeStrategy):
    """
    This strategy uses a special tag value to
    represent a single additional object.
    """
    def check_can_handle(self, value):
       return value is self.wrapped_tagged_value() or \
             (isinstance(value, self.contained_type) and \
             self.unwrap(value) != self.unwrapped_tagged_value())
```

Listing 4.4: `check_can_handle` from the `TaggingStrategy` mixin class

rstrategies provides two supporting mixins outside of the main mixin hierarchy. One of `SafeIndexingMixin` and `UnsafeIndexingMixin` must be added to collections that use strategies with actual underlying storage arrays. These mixins implement two approaches to index checking when accessing elements in a collection. The `SafeIndexingMixin` will raise an error when a `fetch` or `store` operation is attempted with an index that goes beyond the bounds of the collections size. `UnsafeIndexingMixin` will not perform these boundary checks; this can be useful when the boundary checks are implemented on another level by the VM itself.

4.2.1 StrategyFactory

The `StrategyFactory` is the second important part of rstrategies, next to the strategy mixin hierarchy. It is a regular class and handles creation of strategy objects and switching between different strategies. The VM must subclass `StrategyFactory` and provide implementations for two methods. The first one, `instantiate_empty(strategy_type)`, takes a strategy class as input and returns an empty instance of it. This way the VM can have strategy classes with arbitrary constructors; forcing a fixed signature for the `__init__` constructor method would eliminate the need for a `instantiate_empty` method, but make the initialization of strategy objects tricky. The second abstract method of `StrategyFactory` is `instantiate_and_switch(old_strategy, strategy_type, size)`. Its semantics are slightly more complex: the factory must create an instance of `strategy_type` of the given `size`, and replace `old_strategy` with the new instance. In order to switch to the new strategy, the VM has to find the container of `old_strategy`, which can be achieved by using a back reference from the strategy to its containing collection. In the RSqueak VM, for instance, every strategy object has a field named `w_self` pointing back to the collection that is currently using the strategy. The collection object has a reference to the strategy, which has to be replaced in order to implement `instantiate_and_switch`.

In addition implementing two abstract methods, the constructor of `StrategyFactory` must be called to create a functioning factory. The only constructor parameter is the root class of the VMS strategy class hierarchy. The factory will recursively traverse all subclasses of the root class to find all potential strategy classes.

The constructor of `StrategyFactory` is not the only fact enforcing a single class hierarchy for strategies. In order to use different strategies in the same field in collection classes, RPython requires an explicit common superclass to satisfy the static type system.

Optimized Strategy Transitions

In section 3.3 we argued that it is important to create optimized transition routines when switching between certain strategies. Switching between strategies should occur as rarely as possible, but since we cannot eliminate it completely, we try to minimize the performance impact of this overhead. The following explains the support for optimizing strategy transitions built into rstrategies. This implementation requires minimal effort from the VM implementor and works based on a naming convention rather than configuration. For every strategy class the `StrategyFactory` can find, it will add a generated method to the class itself and to the root class of the strategy hierarchy. For a strategy class called `X`, the method `initiate_copy_into` will be generated as shown in listing 4.5. This method implements the visitor pattern by forwarding the call to `copy_from_X` on the parameter object. To make the visitor pattern complete, a default version of `copy_from_X` is generated in the root class, simply invoking the default and non-optimized `copy_from` routine. Now the method `copy_from_X` can be overwritten on any strategy class to optimize the case when switching away from strategy `X`. Implementing the `copy_from_X` method is also the *only* extra step needed to achieve such an optimization. This mechanism has been used two times in the RSqueak VM, eliminating big chunks of machine code from the JIT traces.

```
def initiate_copy_into(self, other_strategy):
    other_strategy.copy_from_X(self)
```

Listing 4.5: Automatically generated visitor method for a strategy class `X`

Selecting the Right Strategy

When an element is added to a collection, the current strategy checks if it can handle and store the incoming value. If this check fails, a deoptimization must be performed by switching to a more general strategy that can hold both the current elements and the new value. In this case the current strategy asks the `StrategyFactory` to handle the deoptimization. The factory object now has to choose the best follow-up strategy. This is a non-trivial decision because the factory does not know about the semantics of the collections in the underlying programming language. To solve this, the factory needs input from the VM implementor in the form of the class decorator `@strategy()`. In Python any function can be used as a class decorator. After the class is created, the decorator function will automatically be invoked with the class as parameter, giving an opportunity to inspect or modify the class. The `@strategy()` decorator is used with a `generalize` parameter, as shown in listing 4.6. The parameter is a list of strategy classes that can be switched to when the decorated strategy must be deoptimized. The decorator generates a `generalized_strategy_for` method on the decorated class that loops over the `generalize` class list and returns the first strategy that is able to handle the incoming element. If the algorithm fails to find a fitting strategy, an Exception is raised. Therefore, the `generalize` list must always contain a generic strategy as the last element, which can store any value. A generic strategy like this is also the only strategy that can have an empty `generalize` list, since it can not be further deoptimized.

```
@rstrategies.strategy(generalize=[
    IntegerOrFloatStrategy, GenericStrategy])
class IntegerStrategy(AbstractStrategy):
    ...
```

Listing 4.6: `strategy` class decorator with `generalize` parameter

The generalization relationship connects the strategies to a *generalization tree*, which is independent of the class hierarchy tree of strategy classes. Figure 4.2 illustrates this using the strategy classes of the RSqueak VM as an example. The `StrategyFactory` can verify that the generalization graph indeed forms a tree, but it has no way of checking it for semantical correctness. Therefore, the `@strategy()` decorator should be used with care.

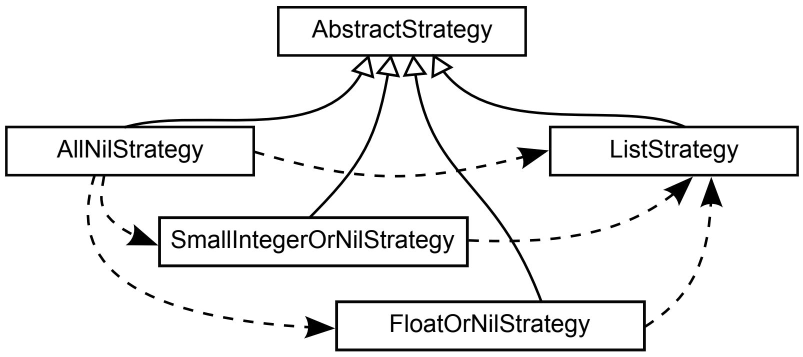

Figure 4.2: RSqueaks strategy classes
The dashed arrows form a secondary hierarchy: the generalization tree.

In some cases VMS also need a routine to find a strategy capable of holding a given value or a list of values. For example, the RSqueak VM loads live objects from an image file at startup. These objects are already filled with values and it is beneficial to load them using the most specialized strategy possible. Another example are language constructs that create a collection initially filled with a given list of values. `StrategyFactory` provides a routine for such cases, called `strategy_type_for`. `strategy_type_for` iterates over a list of objects given as parameter and eliminates all strategies that are not able to hold any of them. From the left over strategies, the most specialized one is returned. This algorithm requires a list of all available strategy classes. As mentioned above, this list is created in the constructor of `StrategyFactory` by traversing all subclasses of a given strategy root class. However, there are two problems with this list: it can include abstract strategy classes that should not be instantiated, and it is not ordered. The `@strategy()` class decorator takes care of the first problem. Aside of generating the `generalized_strategy_for` method it also sets a `_strategy` flag on the strategy class to `True`, marking it as a non-abstract strategy. Only these marked classes are collected into a list of available strategies. After the list is created, it is sorted. In order for `strategy_type_for` to return the most specialized strategy, the more general strategies should be sorted to the end of the list. The sorting criterion is therefore based on the aforementioned generalization tree: the depth of a strategy inside that tree is a good indicator for its level of specialization. The depth of a given strategy is determined by traversing the tree from the strategy up to the root node and counting the number of traversed nodes. If there are multiple paths to the root node, the largest distance is used as depth. For example, in figure 4.2 the `AllNilStrategy` has a generalization depth of 2, `SmallIntegerOrNilStrategy` and `FloatOrNilStrategy` have a depth of 1 and `ListStrategy` is the root node with a depth of 0. The list of strategies is sorted in descending order based on the determined generalization depth.

The implementation presented in this chapter make heavy use of Pythons meta programming capabilities in order to hide implementation details from the user of rstrategies and automate most aspects of storage strategies. RPython allows meta programming as long as it only happens at compile time and the programming structure is frozen at runtime. This rstrategies is fast despite the meta programming, because the generated methods are translated to efficient C code and JITted at runtime, just as regular RPython code.

4.3 Using rstrategies

In the following we explain in more detail how the rstrategies library is used in VM code. The code examples given here are based on the RSqueak VM code base. As described by the previous two chapters, rstrategies consists of two main parts: the strategies mixin class hierarchy and the `StrategyFactory`. The following two sections describe the usage of these two aspects of rstrategies.

4.3.1 Using the Strategy Mixins

In order to use polymorphism, RPython requires one common base class for all strategy classes and rstrategies also follows that paradigm. Listing 4.7 shows the class `AbstractStorageStrategy` which fills that role in the RSqueak VM. The class has no further superclass besides Pythons default `object` class. It defines `StrategyMetaclass` to be its metaclass using Pythons `__metaclass__` mechanism. It also imports two mixins: `AbstractStrategy` and `SafeIndexingMixin`. The former must be imported by the strategy root class and provides basic functionality such as the `init_strategy` method. The latter defines index checking methods that will make sure that elements in RSqueaks strategy classes are accessed correctly. The alternative `UnsafeIndexingMixin` could be used if these checks are already implemented elsewhere. The constructor of `AbstractStorageStrategy` initializes a few attributes and invokes the provided `init_strategy` method. The `strategy_factory` method returns the `StrategyFactory` instance, which is available through the `self.space` field. Finally, the `copy_from_AllNilStrategy` makes use of the strategy switching mechanism described in section 4.2.1. It optimizes the transition from the `AllNilStrategy` to all other strategies by simply doing nothing: all strategies initially contain only `nil` values by default.

```python
class AbstractStorageStrategy(object):
    __metaclass__ = rstrategies.StrategyMetaclass
    import_from_mixin(rstrategies.AbstractStrategy)
    import_from_mixin(rstrategies.SafeIndexingMixin)

    def __init__(self, space, w_self, size):
        self.space = space
        self.w_self = w_self
        self.init_strategy(size)

    def strategy_factory(self):
        return self.space.strategy_factory

    def copy_from_AllNilStrategy(self, all_nil_storage):
        pass # Fields already initialized to nil
```

Listing 4.7: `AbstractStorageStrategy`, based on the RSqueak VM

`AllNilStrategy` is the simplest strategy in RSqueak. It represents the frequent case where all fields in an object contain the `nil` object. Listing 4.8 shows the complete definition of this strategy. The first four lines show the `@strategy()` class decorator with a `generalize` list parameter. It tells the `StorageFactory` that `AllNilStrategy` is a non-abstract strategy class and can be generalized to one of three other strategies when a non-`nil` object is stored into it. The class body itself only imports the `SingleValueStrategy` mixin and defines the `value` method which returns the only object ever contained in this strategy.

```
@rstrategies.strategy(generalize=[
    SmallIntegerOrNilStrategy,
    FloatOrNilStrategy,
    ListStrategy])
class AllNilStrategy(AbstractStorageStrategy):
    import_from_mixin(rstrategies.SingleValueStrategy)
    def value(self): return self.space.w_nil
```

Listing 4.8: `AllNilStrategy`, based on the RSqueak VM

The `SmallIntegerOrNilStrategy`, as shown in listing 4.9, defines a strategy capable to hold integer objects *or* the `nil` object. It can be generalized only to the generic `ListSrategy` and uses the `TaggingStrategy` mixin. The `contained_type` field tells rstrategies that the objects stored here are represented by the `W_SmallInteger` class. The `wrap` and `unwrap` methods convert between machine-level integers and the `W_SmallInteger` representation. The `wrapped_tagged_value` method expresses the fact that this strategy can contain `nil` in addition to integer values. `unwrapped_tagged_value` returns the special tag value used to represent `nil` objects, which is the largest possible integer on the execution platform.

```
@rstrategies.strategy(generalize=[ListStrategy])
class SmallIntegerOrNilStrategy(AbstractStorageStrategy):
    import_from_mixin(rstrategies.TaggingStrategy)
    contained_type = model.W_SmallInteger
    def wrap(self, val): return self.space.wrap_int(val)
    def unwrap(self, w_val): return self.space.unwrap_int(w_val)
    def default_value(self): return self.space.w_nil
    def wrapped_tagged_value(self): return self.space.w_nil
    def unwrapped_tagged_value(self): return constants.MAXINT
```

Listing 4.9: `SmallIntegerOrNilStrategy`, based on the RSqueak VM

4.3.2 Using the `StrategyFactory`

Listing 4.10 shows a slightly simplified version of RSqueaks `StorageFactory`. The first three methods are mandatory for a `StorageFactory` implementation. On line 4, the superclass constructor is invoked with the `AbstractStorageStrategy` class as parameter. This defines the base class of the entire strategy class hierarchy. The `instantiate_empty` method returns an empty instantiation of the given strategy class by passing it the required `self.space` object. `instantiate_and_switch` is more complex: it replaces the old strategy object with a new, non-empty strategy, by following the `w_self` reference stored in every strategy object. The last two methods are facility methods for other parts of the RSqueak VM. `strategy_type_for` is an extension of the same method provided by the default `StrategyFactory` class, dealing specially with weak objects and the `WeakListStrategy`. Finally, `empty_storage` returns an initial strategy for a newly created object. Except for weak objects, all objects start out with the `AllNilStrategy`.

```
1  class StrategyFactory(rstrategies.StrategyFactory):
2      def __init__(self, space):
3          self.space = space
4          rstrategies.StrategyFactory.__init__(self, AbstractStorageStrategy)
5
6      def instantiate_empty(self, strategy_type):
7          return strategy_type(self.space, None, 0)
8
9      def instantiate_and_switch(self, old_strategy, size, strategy_class):
10         w_self = old_strategy.w_self
11         instance = strategy_class(self.space, w_self, size)
12         w_self.store_strategy(instance)
13         return instance
14
15     def strategy_type_for(self, objects, weak=False):
16         if weak:
17             WeakListStrategy
18         return rstrategies.StrategyFactory.strategy_type_for(self, objects)
19
20     def empty_storage(self, w_self, size, weak=False):
21         if weak:
22             return WeakListStrategy(self.space, w_self, size)
23         return AllNilStrategy(self.space, w_self, size)
```

Listing 4.10: `StrategyFactory`, based on the RSqueak VM

4.4 Debugging Storage Strategies

Sometimes a VM developer requires detailed knowledge of what is happening inside the rstrategies system. Automatic optimizations like storage strategies can have an unwanted impact on program performance. In this case, the developer has to debug the library code to find out what is causing the program to slow down and how the situation can be improved.

Debugging an optimization library like rstrategies is hard. Different optimizations can be triggered at any time and the heuristics behind these decisions are not always obvious. In addition, storage access is a very frequent operation which makes breakpoint-based debugging even harder.

A good way to inspect the inner workings of storage strategies is to log storage operations so they can be analyzed after the program is finished. rstrategies provides automatic logging support through the `rstrategies_logging.py` module. Logging can be activated via the `logger` field of `StrategyFactory`. Once activated, strategy switch operations will be logged and printed on the standard output.

A separate script, `rstrategies_logparser.py`, can parse the log format and convert it to useful analysis artifacts like summaries or graphs. An example graphical summary based on the RSqueak VM is given in figure 5.1. Any VM including rstrategies can automatically use these analytical facilities.

5 Evaluation

In this chapter we evaluate the rstrategies library in two general directions: runtime performance and applicability to different programming languages. We evaluate the runtime performance by comparing the results of benchmarks executed with and without storage strategies. We show the generality of the library by demonstrating three programming languages of very different nature that have been successfully adapted to using rstrategies.

5.1 Runtime Performance

In this section we present the results of experiments measuring the execution time of benchmarks to evaluate the performance benefits of storage strategies. The experiments have been conducted on a 64-bit Windows 8.1 machine with 8GB of RAM and an Intel Core i7-2620M CPU with two physical and four virtual cores, clocked at up to 3.40 GHz during single-core execution. During the experiments, no other user applications were running and the experiment process was given increased priority in order to minimize influences from operating system processes. We used the RSqueak VM at commit 1060:7a0ce39abc12 (16.10.2014), Pypy at commit 72481:35fdf446e439 (24.07.2014) and a Squeak 4.5 image to take the measurements.

To evaluate the runtime performance of rstrategies, all experiments were conducted twice. First, all specialized storage strategies have been disabled via a command line switch. In this setup, the `ListStrategy` is used by every object or collection created during the benchmarks. In the second setup, specialized storage strategies were enabled. In the case of the RSqueak VM, these strategies are `AllNilStrategy`, `SmallIntegerOrNilStrategy` and `FloatOrNilStrategy`. We compare the runtime performance of these two experimental setups to evaluate the impact of the specialized storage strategies.

The following list gives short explanations of the seven benchmarks executed during the experiments. These are all common benchmarks based on The Computer Language Benchmarks Game[1].

[1]`http://benchmarksgame.alioth.debian.org/` (August 2014)

AStar The AStar path searching algorithm is used to find a solution through two predefined mazes. This benchmark creates a big graph of interconnected objects and generates lots of non-recursive messages. Two different mazes are solved, called AStar1 and AStar2; the second maze is larger.

BinaryTree Balanced binary trees of a given depth are constructed and then walked, summing up the elements of all leaf-nodes. This benchmark generates many recursive messages.

Blowfish The symmetric-key block cipher Blowfish is used to encrypt and decrypt fixed challenge data. This benchmark is heavy on basic number arithmetic, very few objects are created.

DeltaBlue DeltaBlue is a constraint solver benchmark. Different constraints are created for a number of variables, then the solver finds optimal solutions for the given constraints. This benchmark is heavy on message sends and conditional logic.

NBody The NBody algorithm creates a number of objects with an assigned movement vector and mass, then performs a round-based simulation of their movement and interactions using Newton's laws of motion. This benchmark balances basic arithmetics and message sends.

Richards The Richards benchmarks simulates the task scheduler of an operating system kernel. Multiple classes of tasks can be created with different priorities, then the tasks are scheduled until every task has finished. The main workload of this benchmark consists of message sends and conditional logic.

SplayTree The SplayTree benchmark creates a splay tree of a fixed size and then performs a number of tree modifications by removing the greatest key node in the tree. This benchmark creates and modifies a big interconnected graph of objects.

For all experiments, the entire benchmark suite is executed within the same vm instance. This creates a more realistic setup than many separate processes, since a real-world application is likely to be long-running and to perform multiple different tasks.

5.1.1 Results

The execution time of the benchmarks was measured by recording timestamps at the beginning and end of every benchmark. Every benchmark was executed 50 times to achieve a satisfactory significance level. From the 50 measurements taken for every benchmark, the first n are excluded from the analysis to eliminate the impact of the jit warmup time. The number n is determined manually for every benchmark, since the time it takes to reach steady performance differs from

Benchmark	Excluded Measurements	Analyzed Measurements
AStar1	14	36
AStar2	14	36
BinaryTree	3	47
BlowfishDecryption	13	37
BlowfishEncryption	16	34
DeltaBlue	8	42
NBody	3	47
Richards	10	40
SplayTree	12	38

Table 5.1: Excluded and analyzed measurements for the different benchmarks

one benchmark to another. Table 5.1 lists the number of excluded measurements for every benchmark.

The remaining measurements have been summarized by means of confidence intervals using a 5% significance level. The confidence intervals comparing the two VMS have been computed following the procedure proposed by Kalibera et al. [18]. The confidence intervals summarizing the results of the individual benchmarks show the standard error of the population, assuming normal distribution. Table 5.2 summarizes the results, each line represents one of the benchmarks. The table is divided in three columns, each summarizing different results for the respective benchmark. The first column shows the results for the RSqueak VM *without* storage strategies, i.e. with the rstrategies library disabled. The second column shows the results for the RSqueak VM with storage strategies enabled. The numbers in the first two columns are given in milliseconds. The third column shows the relative performance change from the first to the second section. A positive number indicates increased performance, meaning that the benchmark was executed in less time.

The AStar and SplayTree benchmarks showed considerable increase in performance, while Blowfish and NBody showed a minor decrease. The other three benchmarks showed minor increases between 1% and 7%. Since the means and standard errors shown in the first two columns differ quite substantially, it would not be meaningful to construct a single confidence interval from all benchmarks. Instead, we observe that storage strategies provide high performance benefits for certain problems, while the potential performance penalty remains very low. The average half-width of the confidence intervals is 1.99%, which is reasonably low for a significance level of 5%.

Benchmark	Without Strategies	With Strategies	Performance Change
AStar1	80.64 ms ±1.92	55.81 ms ±2.61	+46.93 % ±6.58
AStar2	351.17 ms ±8.15	288.81 ms ±2.68	+21.21 % ±2.25
BinaryTree	187.47 ms ±1.00	174.94 ms ±1.77	+7.26 % ±1.01
BlowfishDecryption	422.16 ms ±2.05	429.16 ms ±2.49	−1.62 % ±0.64
BlowfishEncryption	423.85 ms ±2.00	427.15 ms ±2.47	−0.76 % ±0.63
DeltaBlue	99.86 ms ±2.24	98.31 ms ±2.16	+1.57 % ±2.62
NBody	271.38 ms ±2.15	274.09 ms ±2.16	−0.98 % ±0.94
Richards	165.97 ms ±2.80	162.95 ms ±4.14	+2.11 % ±2.29
SplayTree	781.16 ms ±2.25	469.92 ms ±2.89	+66.28 % ±0.97

Table 5.2: Performance measurements for different benchmarks
From left to right: RSqueak VM without specialized storage strategies, RSqueak VM with specialized storage strategies, relative performance change between the first two. Measurements given as 95 % confidence intervals.

The two benchmarks that benefit most, AStar and SplayTree, both create large graphs of objects. Therefore, it is not surprising that they benefit a lot from an allocation removal optimization. The benchmarks with slight performance losses, Blowfish and NBody, are the two benchmarks with the highest focus on arithmetical operations. Since there are not many object allocations happening here, we can measure the overhead of storage strategies as a slight performance drop.

5.2 Reusability of Implementation

An important aspect of the rstrategies library is its independence of the language executed by the client VM. Not only is the programming language not significant – the *class* or *family* of programming languages does not matter either. In order to evaluate this, we introduced rstrategies into three different RPython based VMS by extending certain datatypes to use storage strategies. In the RSqueak VM, every single object uses storage strategies. In the Ruby VM Topaz, we added rstrategies to the `array` datatype. Finally, in the Pycket VM, the `vector` datatype has been enhanced by rstrategies.

While is is not possible to prove that rstrategies is applicable to any arbitrary programming language, we believe that the three selected examples cover a wide range of programming languages and families. Squeak and Ruby are both object-oriented and imperative languages; while Ruby is a scripting language, Squeak is an image-based language that loads an entire live programming environment before executing the first bytecode. Racket belongs to the Lisp/Scheme programming language family. It is a multi-paradigm language including object-oriented features, but the syntax, control flow and main data structures are typical for a functional programming language. We were able to integrate rstrategies in the RPython based VMS for all three languages in largely the same way.

To extend this evaluation, we use the two VMS RSqueak and Topaz to demonstrate how rstrategies works in the context of different programming languages. The third VM, Pycket, was not mature enough to execute Racket code of sufficient complexity at the time of this evaluation. Pycket, however, still contains a full implementation of the rstrategies optimizations and we observed good performance results in dedicated micro benchmarks.

rstrategies includes a logging facility which, if enabled, outputs a log of all strategy related operations such as the creation of collections or switching between different strategies. A log parsing module of rstrategies can convert these logs into different formats, including a visualization of the transitions between different storage strategies. We used this technique to create such transition graphs for RSqueak and Topaz, while executing different benchmark suites. For RSqueak we reused the benchmark suite from the previous chapter. For Topaz we used a benchmark suite consisting of the 5 benchmarks binary-trees, dhrystone, mandelbrot, revcomp and richards. We have not selected two equal benchmark suites since we are not conducting a performance comparison between the two VMS.

Figure 5.1 shows the resulting transition diagram for the RSqueak VM. Objects are created at two different times: when loading the image and after the image is loaded. From there the objects transition to their initial strategy. Objects created at image loading time are directly instantiated with the best-fitting strategy, while objects created later always start off with the `AllNilStrategy`. The `WeakListStrategy` is a special case: weak objects never transition to any other strategy after creation. From the `AllNilStrategy`, objects transition to one of the specialized `SmallIntegerOrNilStrategy` or `FloatOrNilStrategy`, or directly to the generic `ListStrategy`. The percentage of objects `remaining` in any of the specialized strategies is a good indicator for how well the storage strategies perform. In the case of RSqueak, approximately 25% of all objects never leave the `AllNilStrategy`. These objects are mainly `MethodContext` objects which are created automatically upon message sends. 97% and 90% of all objects arriving at the `SmallIntegerOrNilStrategy` or `FloatOrNilStrategy`, respectively, actually stay in these specialized strategies. These numbers indicate a high success rate of the heuristics behind choosing the storage strategies.

The node at the bottom labeled *Other* summarizes a few rarely used strategies which are special to RSqueak.

Figure 5.2 shows an example transition graph for the Topaz VM. It has less nodes than the RSqueak diagram because in Topaz strategies have only been added to the `array` datatype, not to every single object. The `IntStrategy` has a 100% success rate, which is supposedly due to the nature of the benchmarks executed while capturing this data. The `EmptyStrategy` also has a rather high success rate, while the `FloatStrategy` only works one out of three times. Compared to the RSqueak VM, these numbers look less conclusive and more artificial. The main reason is the different nature of the two VMS: RSqueak loads an entire programming environment, including things like a process scheduler and user interrupts. Therefore, the VM is executing some unrelated code even while running these benchmarks. Topaz, on the other hand, evaluates pure Ruby scripts, which results in a more deterministic and uncluttered execution. Unfortunately, Topaz is not yet able to execute a larger Ruby application to provide a less artificial picture.

The two transition diagrams show that rstrategies has been successfully integrated in two different VMS, namely RSqueak and Topaz. The same has been done with the Pycket VM. Of course this does not prove that rstrategies can be used in any number of arbitrary RPython VMS. Nevertheless, we think that the provided evidence supports our claim about the generic nature of the rstrategies library.

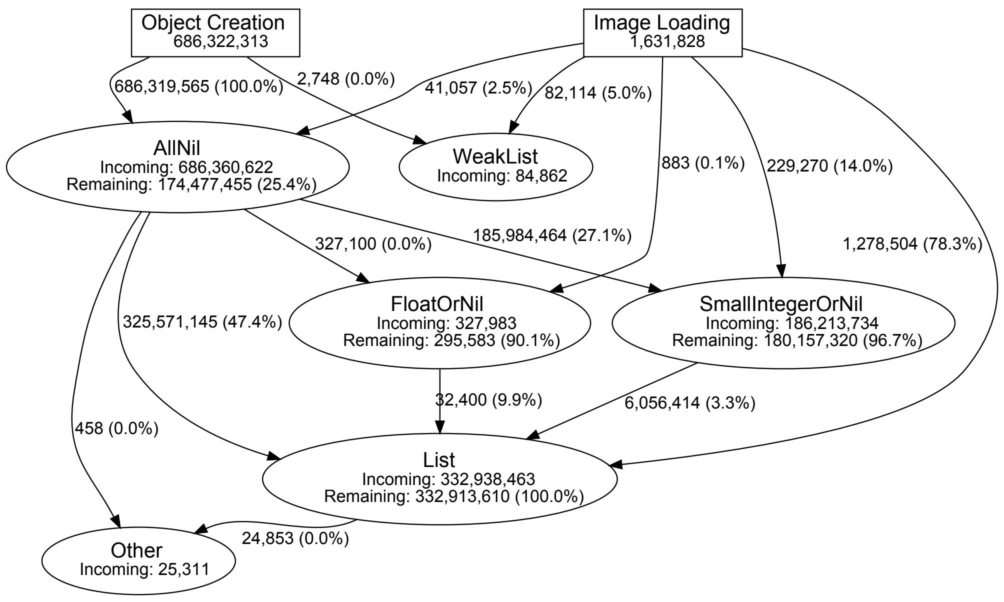

Figure 5.1: Strategy transitions of the RSqueak VM executing benchmarks

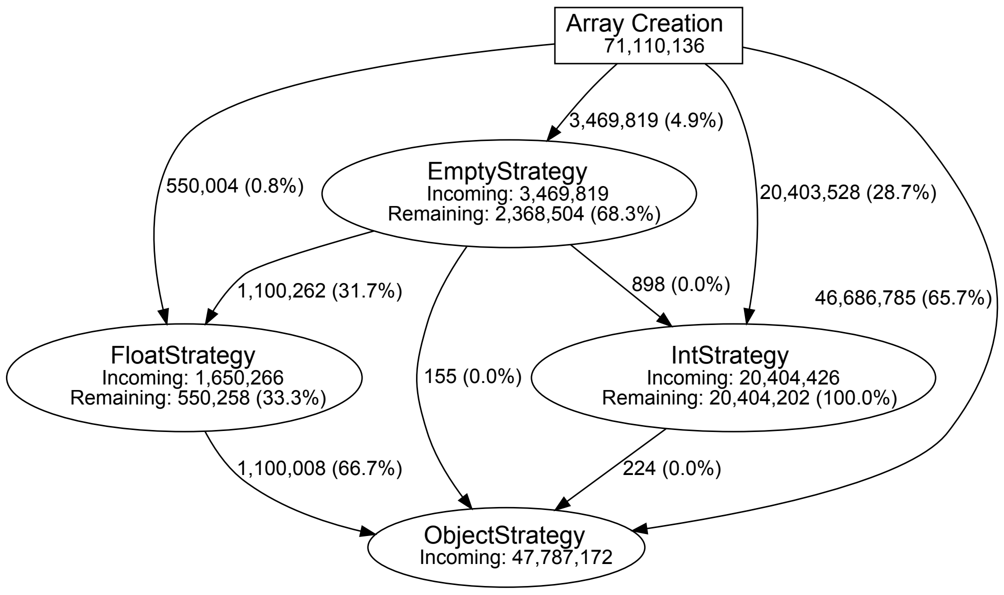

Figure 5.2: Strategy transitions of the Topaz VM executing benchmarks

6 Related Work

RPython based VMS are not the only ones featuring optimization technologies like a JIT or storage strategies. Because of the advantages such modern optimizations provide for dynamic language runtimes, they are quickly adapted and reused in different projects. In this chapter we present a selection of current VM projects, their approaches to optimizing dynamic programming languages and how they differ from RPython and the rstrategies library.

6.1 Graal and Truffle

The Graal/Truffle project is a VM building framework similar to RPython in its main design goal. An interpreter written in a statically typed programming language is used as the source artifact to automatically produce an efficient VM including a JIT.

Graal [20] is an experimental extension to the HotSpot VM which exposes VM internal mechanisms to the executed program via a Java API. Namely, HotSpots method based JIT compiler can be controlled from the client program. Truffle [23] is a generic, self-optimizing AST interpreter that makes use of the interface provided by Graal. In order to implement a VM using Graal/Truffle, the program code must be parsed into a syntax tree and passed to Truffle for execution. Truffle will dynamically modify the AST using techniques like partial evaluation and inlining to create compilation units for heavily used code paths, which are then converted to machine code by the Graal JIT. This approach effectively uses a method JIT compiler, but is actually similar to RPythons meta tracing JIT.

The main difference between RPython and Graal/Truffle, aside from the underlying infrastructure, lies in the flexibility for the VM programmer. Graal/Truffle requires the VM implementor to use an AST, while the RPython toolchain accepts arbitrary Python programs as long as they satisfy the static typechecking.

The project JRubyTruffle[1] is a Ruby VM written in Java which uses the Graal/Truffle infrastructure. JRubyTruffle includes storage strategy optimizations [22], but not in form of a reusable library like rstrategies.

6.2 LLVM

LLVM [19] is a compiler infrastructure framework mainly intended for building static compilers. LLVM defines an intermediate representation (IR) bytecode that can be passed to various back ends to be converted to optimized native machine code. A typical three-tier static compiler only needs to implement a front end that produces the IR. LLVM also has JIT capabilities allowing to transform the bytecode to native machine code at runtime. Since LLVM is a C++ framework, the JIT functionality has to be invoked explicitly, in contrast to RPython. LLVMs JIT is also more related to a method JIT than a tracing JIT like generated by RPython.

Due to its good design and modularity, LLVM is used in many projects researching compilation techniques. Böhm et al. have used LLVM to create a multi-threaded dynamic binary translator, which acts like a JIT between different native machine bytecodes [2]. LLVM can also be used for different tasks, like dynamic decoding of MPEG Dynamic Video Coding data [15].

6.3 TraceMonkey

TraceMonkey [12] is a JIT compiler for JavaScript. It used to be part of Mozilla's SpiderMonkey VM, before being replaced by JägerMonkey and then IonMonkey. The tracing mechanism of TraceMonkey is very similar to RPythons JIT. TraceMonkey includes architectural parts which are independent of the executed language, but we do not know of any cases where TraceMonkey code has been reused to implement other VMS.

6.4 V8

Google's Javascript engine V8[2] has a JIT compiler with similar performance optimization techniques to RPython and VMS written in RPython. The main concept behind optimizing highly dynamic Javascript code is the notion of *hidden classes*. Even though inheritance in Javascript is based on prototype

[1]http://www.chrisseaton.com/rubytruffle/ (August 2014)
[2]https://code.google.com/p/v8/ (August 2014)

objects instead of classes, the V8 VM maintains a hidden class for every object. The class is changed based on the properties which are stored inside an object or array. This concept can be used to achieve similar goals as storage strategies, but applies for both arrays and regular objects.

In order to write efficient code for the V8 engine, properties should rarely be added or deleted from objects because this causes the hidden class to be changed. When the hidden class of a Javascript object is changed due to an added property, the VM might have to reallocate and copy the objects storage in order to make room for the new property. The same goes for switching storage strategies, which leads to a deoptimization procedure.

In the context of arrays, V8's hidden classes implement the same basic mechanism as storage strategies. As long as an array contains only floats, the unboxed float data will be stored in consecutive memory.

In contrast to RPython, V8 uses *tagging* to store 31-bit integer numbers directly inside pointers. RPython and rstrategies do not follow this approach due to the increased complexity and because the tracing JIT does a good job in eliminating integer box objects.

7 Conclusion

In this thesis we presented rstrategies, a library written in RPython for reusing the storage strategies optimization across different virtual machines, regardless of what programming language they implement. Storage strategies is an example of an allocation removal optimization: the VM transparently inlines the state of certain objects into a collection, instead of allocating the objects on the heap. By doing this, the VM avoids “real” allocations and saves memory, accesses to memory and processing time.

RPython is a statically typed subset of the object-oriented scripting language Python, and a VM written in RPython can be automatically translated to an efficient C version including an optimizing JIT compiler. Even though the RPython toolchain automatically performs many optimizations during the translation process, storage strategies is not one of them. Instead, every VM includes a custom implementation of the storage strategies optimization, duplicating code across related systems and preventing improvements in storage strategies to migrate from one VM to another.

rstrategies can reduce the complexity of a VM codebase: the VM only has to import and configure the functionality of rstrategies.

In order to be reusable, rstrategies provides a flexible API based on mixins. Different mixins can be instantiated in VM specific classes and the VM can specialize or overwrite the default functionality where necessary. To control the mechanisms behind instantiating and switching different storage strategies, the VM uses a custom subclass of `StorageFactory`. In order to provide a minimal, yet powerful and generic set of functionality, we analyzed the requirements of 10 modern, VM-based programming languages. We also analyzed potential extensions of rstrategies and explained the deliberate limitations of the library.

We evaluated our library implementation by measuring the performance impact of the rstrategies optimizations. For that we used the RSqueak VM to execute a suite of benchmarks both with and without storage strategies enabled. We observed that in suitable benchmarks rstrategies can achieve a speedup of up to 66%, while the performance penalty in unoptimized situations did not exceed 1.62%. As expected, benchmarks that cause many object allocations and memory accesses can be optimized well, while arithmetic-heavy algorithms tend to suffer a small slowdown.

To demonstrate the reusability of our implementation we added **rstrategies** support to 3 existing RPython VMS: RSqueak, Topaz and Pycket. These VMS implement Squeak/Smalltalk – an image-based pure object-oriented language, Ruby – an object-oriented scripting language and Racket – a functional programming language, respectively. We showed the operation of **rstrategies** in RSqueak and Topaz by creating *strategy transition diagrams*. These diagrams were automatically created by the logging facility included in **rstrategies**.

To the best of our knowledge, **rstrategies** is the first attempt to generalize the storage strategies optimization and make it available to multiple VMS.

Future work on the **rstrategies** library is two-fold. First, we plan to extend **rstrategies** and experiment with other, possibly new aspects of storage strategies. The main aspect we want to consider are strategies for optimizing dictionaries in a similar way as arrays and lists. Languages with primitive dictionary types, like Ruby or Python, could benefit from such optimizations. Second, we plan to add **rstrategies** support in more RPython based VMS like the Prolog VM Pyrolog. With this we want to find out whether the storage strategies optimization is useful in different language families like declarative and logical programming.

In conclusion, we consider **rstrategies** a useful addition to RPythons **rlib**, and work is underway to release it at least with RSqueak, Topaz and Pycket.

Bibliography

[1] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D Matsakis. "RPython: a step towards reconciling dynamically and statically typed OO languages". In: *Proceedings of the 2007 symposium on Dynamic languages.* ACM. 2007, pp. 53–64.

[2] Igor Böhm, Tobias JK Edler von Koch, Stephen C Kyle, Björn Franke, and Nigel Topham. "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator". In: *ACM SIGPLAN Notices.* Vol. 46. 6. ACM. 2011, pp. 74–85.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. "Allocation removal by partial evaluation in a tracing JIT". In: *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM. 2011, pp. 43–52.

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. "Tracing the meta-level: PyPy's tracing JIT compiler". In: *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* ACM. 2009, pp. 18–25.

[5] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. "Storage strategies for collections in dynamically typed languages". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications.* ACM. 2013, pp. 167–182.

[6] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. "Back to the future in one week-implementing a Smalltalk VM in PyPy". In: *Self-Sustaining Systems.* Springer, 2008, pp. 123–139.

[7] Gilad Bracha and William Cook. "Mixin-based inheritance". In: *ACM SIGPLAN Notices* 25.10 (1990), pp. 303–311.

[8] Craig Chambers. "The design and implementation of the self compiler, an optimizing compiler for object-oriented programming languages". PhD thesis. Stanford University, 1992.

[9] Craig Chambers, David Ungar, and Elgin Lee. “An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes”. In: *ACM SIGPLAN Notices*. Vol. 24. 10. ACM. 1989, pp. 49–70.

[10] Tim Felgentreff. “Ruby Topaz”. In: *wroc_love.rb*. Mar. 2013.

[11] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. “Classes and mixins”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 171–183.

[12] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. “Trace-based just-in-time type specialization for dynamic languages”. In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 465–478.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[14] Benjamin Goldberg and Young Gil Park. “Higher order escape analysis: optimizing stack allocation in functional program implementations”. In: *ESOP’90*. Springer, 1990, pp. 152–160.

[15] Jérôme Gorin, Matthieu Wipliez, Françoise Prêteux, and Mickaël Raulet. “LLVM-based and scalable MPEG-RVC decoder”. In: *Journal of Real-Time Image Processing* 6.1 (2011), pp. 59–70.

[16] David Gudeman. *Representing type information in dynamically typed languages*. 1995.

[17] IEEE. *IEEE 754: Standard for Binary Floating-Point Arithmetic*. Aug. 2014. URL: http://grouper.ieee.org/groups/754.

[18] Tomas Kalibera and Richard Jones. “Rigorous benchmarking in reasonable time”. In: *ACM SIGPLAN Notices*. Vol. 48. 11. ACM. 2013, pp. 63–74.

[19] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[20] Oracle. *OpenJDK: Graal project*. Aug. 2014. URL: http://openjdk.java.net/projects/graal/.

[21] Armin Rigo and Samuele Pedroni. “PyPy’s approach to virtual machine construction”. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM. 2006, pp. 944–953.

[22] Chris Seaton. *Optimising Small Data Structures in JRuby+Truffle.* Aug. 2014. URL: `http://www.chrisseaton.com/rubytruffle/small-data-structures/`.

[23] Christian Wimmer and Thomas Würthinger. “Truffle: a self-optimizing runtime system”. In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity.* ACM. 2012, pp. 13–14.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Berlin, den 5. Dezember 2014

Anton Gulenko