INSTITUT FÜR INFORMATIK
Softwaretechnik und
Programmiersprachen

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Memory Optimizations for Data Types in Dynamic Languages

Lukas Diekmann

Masterarbeit

Beginn der Arbeit:	23. August 2011
Abgabe der Arbeit:	23. Februar 2012
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Michael Schöttner

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 23. Februar 2012

_____

Lukas Diekmann

Abstract

Dynamic languages are often slower and use more memory than statically typed languages. The reason for this is the dynamic typing. This thesis shows that by optimizing data types for collections like lists, sets and dictionaries as well as data types on instance variables, memory usage can be reduced by 10% to 50% and execution time by 5% to 90%.

Contents

1 Introduction

Using just-in-time compilers in dynamic languages is a good way to type-specialize data types that reside on the stack, like local variables. The goal of this thesis is to optimize data that is located on the heap. Two major problems of dynamic languages are speed and memory. Due to their dynamic typing these languages are harder to make fast than statically typed ones. Also, due to dynamic typing an additional level of abstraction in the memory layout is necessary which wraps all data types into objects. This costs a lot of memory. A solution for this problem is to remove that boxing in some cases. This solution is based on the assumptions that a lot of data types are stored in collections such as lists, sets and dictionaries and that storing mixed types is unlikely for them. Removing the boxing does not only save up the memory that is used by the wrapper object, but also gives the JIT information about the data types during runtime which it can use to optimize generated code.

This thesis proposes to remove the boxing for primitive data stored in containers such as lists, sets and dictionaries. This is possible when such a data structure stores only elements of the same primitive type. This happens often in practical programs. A related optimization can be performed on instances.

The contributions of this thesis are:

- Analysis of memory usage of Python programs
- Memory optimization for data types in lists, sets and dictionaries that store primitive data types only
- Faster interaction between data types in those collections
- Speed optimizations for operations on collections by using the newly gained type information about their content
- Memory optimization for instance variables storing primitive data types

The thesis is structured as follows: Section 2 gives an introduction to dynamic languages, Python and the PyPy project. It also explains how data types are implemented in dynamic languages and how this affects memory usage. Section 3 presents the results of the memory analysis of several Python programs and explains how this was done and which programs were used. In Section 4 it is explained how collections such as lists, sets and dictionaries can be optimized by implementing the proposed solutions in PyPy. It is also shown how operations can be made faster by using the information that are gained from using unwrapped data types and how this influences the JIT. Section 5 explains how instance variables are stored in PyPy and how they can be optimized by removing the boxing and using integer tagging. Section 6 presents results of how the optimizations reduce memory usage and execution time.

2 Background

This thesis mostly treats dynamic languages, their behaviour and implementation. Everything concerning the implementation of the discussed techniques is done within the PyPy project. Therefore, it is necessary to explain some terms and some of the used technologies.

2.1 Dynamic Languages

Dynamic Languages are a class of high-level languages. The difference to statically typed languages is that most of their behaviour is changeable at runtime. This results in some major properties:

- *dynamic typing*: Types are not declared in the sourcecode, but are attached to the values during runtime.
- *reflection*: In many of these languages it is possible to inspect and even change the running program from within itself.
- *late binding*: An object that is referenced by a variable can only be determined at runtime. This also means that the same variables may reference to different types during execution.

In practice dynamic languages also have other properties. For instance, many of them come with an *interactive* console where parts of a program can be written and immediately be executed. Furthermore, they are *garbage-collected* and *interpreted*. Another important property that can be found in many dynamic languages is that *everything is an object*. This means that all objects are manipulated the same way. This applies to instances, lists, modules, classes, functions, strings and even primitive datatypes as numbers and booleans.

A common implementation technique for dynamic languages is interpretation. This can be done by compiling the user program into a set of bytecode instructions and running these bytecodes in an emulator or by builing and iterating over an abstract syntax tree (AST). But, since iterating over the AST is very slow, bytecode compilation is the preferred method. However, interpretation is not a necessary property. Dynamic languages could also be compiled to machine code. The problem with compilation is that due to dynamic typing the source code of dynamic languages does not contain enough information to make the machine code efficient. To compensate the slow interpretation, today many dynamic language implementations use just-in-time compilation. However, whereas writing an interpreter for a dynamic languages is relatively easy it is quite complicated to write a JIT compiler. Some popular examples for dynamic languages are PHP [Gro12], JavaScript [Int99], Python [VR+94], Perl [WCO00], Ruby [FM08] and Tcl [Tea12].

2.2 Python

Python was created by Guido van Rossum at Stichting Mathematisch Centrum Netherlands in the early 1990s [Lut96]. It is a powerful dynamic programming language that is used in a wide variety of application domains. Python has a very clear and readable syntax and offers features like intuitive object orientation, strong introspection, garbage collection and high-level dynamic data types. Furthermore it has a huge amount of standard libraries for almost any task. The reference implementation is CPython and is written in C. The sourcecode is compiled into bytecode and interpreted via a bytecode VM.

2.3 The PyPy Project

PyPy was started in 2003 as a Python interpreter written in a high-level language, Python. There are several advantages of writing a VM in a high-level language. Back then the goal of PyPy was the implementation of a full featured, customizable and fast implementation of Python. By using a high-level language it should be easy to experiment with new language features and implementation techniques and introduce them without changing too much. For example, it should be relatively easy to try out different garbage collectors. Gradually, PyPy evolved and became a common translation framework for dynamic languages. The goals changed too and today PyPy aims to provide an easy way to write flexible implementations for different dynamic languages that can be kept free of low-level details like memory management, object layout and threading model [RP06]. By using PyPy's translation toolchain, the implementations can be translated into various target environments while adding those low-level details as well as garbage collection or a tracing JIT in the process.

However, implementing virtual machines in PyPy does not allow the use of the full Python language. Instead a "compromise between expressivity and the need to statically infer enough type information to generate efficient code" [AACM07] had to be found. Thus, the language RPython was developed. RPython stands for "Restricted Python" and is a subset of Python. It is still valid Python code but it does not allow dynamic typing (i.e. types don't have to be written, but are inferred). Code written in RPython can be translated using PyPy's translation framework. It converts RPython programs into an efficient low-level version for a specific target platform. Currently supported are C/Posix, CLI and JVM. For the translation to the target platform several steps are needed, where each step reduces the level of abstraction until conversion to source code is done which can then be compiled. These steps include conversion to a control flow graph, performing type inference and optional optimizations like inlining.

Although PyPy changed over the years one major task is still the development of the Python interpreter. Of course it is written in RPython, too, and then translated to C using PyPy's translation toolchain. It implements the full Python language and is currently compatible to CPython up to version 2.7.2. The interpreter consists of three components: A bytecode compiler, a bytecode evaluator and the object space. The bytecode compiler reads the source code of a user application and produces Python code objects from it. The bytecode evaluator then interprets these objects. The third

component of the interpreter, the standard object space, can be thought of as a library of builtin types [Doc12] and is responsible for creating and manipulating objects that are seen by the application. It provides a complete set of interpreter level classes that implement later application level objects (e.g. strings, integers, lists, dictionaries, sets, etc.). All of those application level objects have one parent at the interpreter level, the abstract class `W_Object`. As in Python in PyPy everything is an object too, so we need this parent class so the language can be dynamically typed. Examples for objects subclassing `W_Object` are `W_IntObject`, `W_StringObject`, `W_ListObject`.

Since interpretation is very slow compared to compilation, PyPy uses a tracing JIT to improve performance. Tracing JITs have been initally explored by the Dynamo Project [BDB00] and soon proved to be a relatively easy way to implement just-in-time-compilers. Tracing JITs are built upon the assumption that programs typically spent most of their time in loops and these loops are very likely to take the same paths. To speed up execution the interpreter has a profiling phase during interpretation where it searches for frequently executed loops, called hot loops. On finding such a loop a special tracing mode is entered that logs the operations the interpreter does and generates machine code from it. This machine code can then be executed every time the loop runs again which could be immediately in the next iteration step or any time in the future.

2.4 Memory Effects of Dynamic Languages

One of the effects of dynamic typing in dynamic languages is that every variable can store objects of arbitrary types. That means that all objects need a uniform representation in memory, even primitive ones like integers and floats. This makes it possible that all operations can be equally performed on all objects. Of course this still means that attempting to perform such an operation can produce a runtime error. For instance, have a look at the following Python code:

```
if cond:
    x = 5
else:
    x = "abc"
y = x * 3
```

The multiplication can act on either an int or a string. That means that it is necessary for both ints and strings to be represented by a small object on the heap. This representation of primitives as small heap objects is called boxing or wrapping[1].

Unfortunately, boxing primitive datatypes like integers, floats and strings, costs a lot of memory. For example an integer in C on a 32-bit machine has the size of 4 bytes. In CPython it is 12 bytes which is three times as big. The problem is that the boxing object needs memory, too, as can be seen in Figure 1.

If we have a collection of integers, for instance stored in a list, this becomes even worse. Because the integer is wrapped into an object it can not be stored directly on the list, as

[1] Of course another approach would be to use integer tagging to store the integer itself. [Gud93]

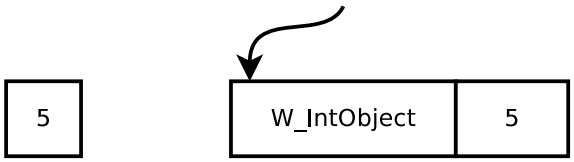

Figure 1: C vs. CPython integer

it would be in C, but needs to be referenced by a pointer. This indirection would cost additional 4 bytes. Figure 2 shows the need for indirections when implementing a list in a dynamic language using the example of PyPy. In the layout every box corresponds to one word of memory. The boxes "GC" come from CPython's reference count field of its garbage collector, in PyPy the garbage collector needs a word for various status flags.

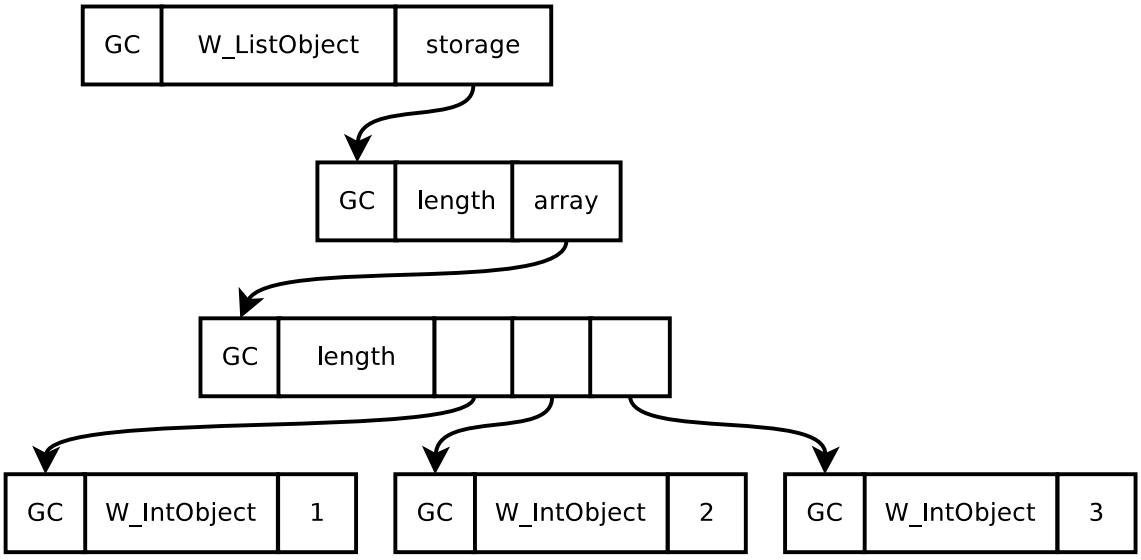

Figure 2: Memory layout of a list containing integers

So, in total there are *extra* 12 bytes per element for a list of integers in CPython compared to an array in C. Let's say we have a list of 1.000.000 integers. Then the same list would need 11.44 MiB more memory in CPython than it would in C.

3 Memory Research in PyPy

To find out which objects use the most memory the following programs were analyzed and the memory usage during runtime was observed. These programs are later also used as benchmarks to measure the quality of the optimizations presented in this thesis. To achieve some meaningful results, important properties of those benchmarks are that they run in a reasonable amount of time, are real applications and, most important, consume a sizable amount of memory (6 - 1000 MiB). In addition a few micro benchmarks were written (tagged with "*") to show best/worst case scenarios. The results were obtained by running the programs until a certain point was reached were they use the most memory. Then different techniques were used to measure the memory and the used data types. So this data only represents memory usage during a specific point of time and was not gathered by monitoring the whole execution to search for allocations of the different types.

3.1 Benchmarks

Here is a description of all benchmarks and how they were used to analyze the memory they consume.

- **NetworkX**: "NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks" [Dev12a]. Memory usage is measured after creating a Barabási and Albert graph with 5000 nodes and a degree of 100.
- **scapy**: "Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies. [...] It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery [...].[Log12] The benchmark uses Scapy to sniff an offline dump of captured packets ($\sim$ 800KB).
- **pyexcelerator** is a Python library for "generating Excel 97+ files with Python, importing Excel 95+ files, support for UNICODE in Excel files, using variety of formatting features and printing options, Excel files and OLE2 compound files dumper." [Dev12c] For the benchmark a new Excel sheet is created with 6000 rows and 200 columns of data.
- **PyDbLite**: "PyDBLite is a fast, pure-Python, untyped, in-memory database engine using Python syntax to manage data, instead of SQL." [Que12]. For the benchmark a table with the columns 'word', 'row', 'pos' and 'filename' is created. Then two books ('Faust1' [Goe00a] and 'Faust2' [Goe00b]) are read and for each word the word itself, the row, the position and the filename are inserted into the database.
- **pypy-interp**: This benchmark uses PyPy to interpret a program creating a list and adding 5000 tuples containing integers, strings, floats, dictionaries and user-defined objects to that list.

- **pypy-translate** In this benchmark PyPy's *translation toolchain* is used to translate an RPython version of the common Richards benchmark to C.

- **whoosh**: "Whoosh is a fast, featureful full-text indexing and searching library implemented in pure Python. Programmers can use it to easily add search functionality to their applications and websites. Every part of how Whoosh works can be extended or replaced to meet your needs exactly" [Dev12b]. In the benchmark whoosh is used to index Goethes Faust 1 [Goe00a] and Faust 2 [Goe00b].

- **Feedparser** is, as the name suggests, a program for reading RSS feeds. The benchmark measures after parsing $\sim$ 8.5MB of (locally stored) small RSS feeds.

- **nltk-wordassoc**: Uses the Natural Language Toolkit [Nlt12] to do some text analysis like counting all nouns that occur ahead of other nouns. This benchmark was inspired by examples in [Mad07].

- **Disaster**: "Disaster aims to do simultaneous and morphological disambiguation and syntactic chunking of Slavic languages (currently Polish) using Machine Learning techniques" [RP10]. The benchmark measures after training a source.

- **Bazaar** is a version control system like Git or Mercurial. The benchmark, initializes a new Bazzar repository and adds a Python project consisting of over 700 files with the total size of $\sim$ 57 MB. The benchmark hooks after packing the repository.

- **Multiwords**: "Multiwords implements the LocalMaxs algorithm for extracting multiword units (MWUs) from plain text", described in [SL99]. The benchmark runs the multiwords algorithm on top of Goethes 'Faust: Der Tragoedie zweiter Teil' [Goe00b].

- **orm**: This benchmark, created by [Bay12], uses SQLAlchemy to create an object-relational mapping to store some data into an SQLite Database.

- **slowsets** is a small Python program that uses sets to find all combinations of the letters A-Z.

- **findprimes** is a small Python program that uses generators to find prime numbers.

- **invindex** implements an inverted index algorithm in Python as described in [Cod12] and analyzes Goethe's Faust1 and Faust2 [Goe00a, Goe00b].

- **liststrategy*** creates three different lists with 1000000 elements each. The first one containing only integers, the second one containing strings and the last containing integers but instead of the others was created by using Pythons *range* method.

- **setstrategy*** is equivalent to the *liststrategy* benchmark, storing a huge number of integers and strings in sets.

3.2 Analysing memory usage

As explained in 2.4 especially interesting is the memory usage of collections such as lists, sets and dictionaries. To measure this a program was written which executes each benchmark and analyzes all objects that are found in the memory during the execution of that benchmark. The easiest way to do this is by using the garbage collector to iterate over all objects it manages during runtime. A list of all objects in memory can be received by importing the garbage collector module (`import gc`) and calling `gc.get_objects`. The size of an object can be measured by using the system module and it's method `sys.getsizeof(obj)`. However, this will only return the size needed by the object itself. It will not measure the size of other objects it points to. To calculate the total size of a collection and all its objects we need to follow the collections referents, measure their sizes and add this to the total value.

3.3 Results

In this section, several diagrams are presented, showing different collections storing different data types. Specifically, it was measured how many objects are stored, the size they take up in memory and their percentage to the total heap size.

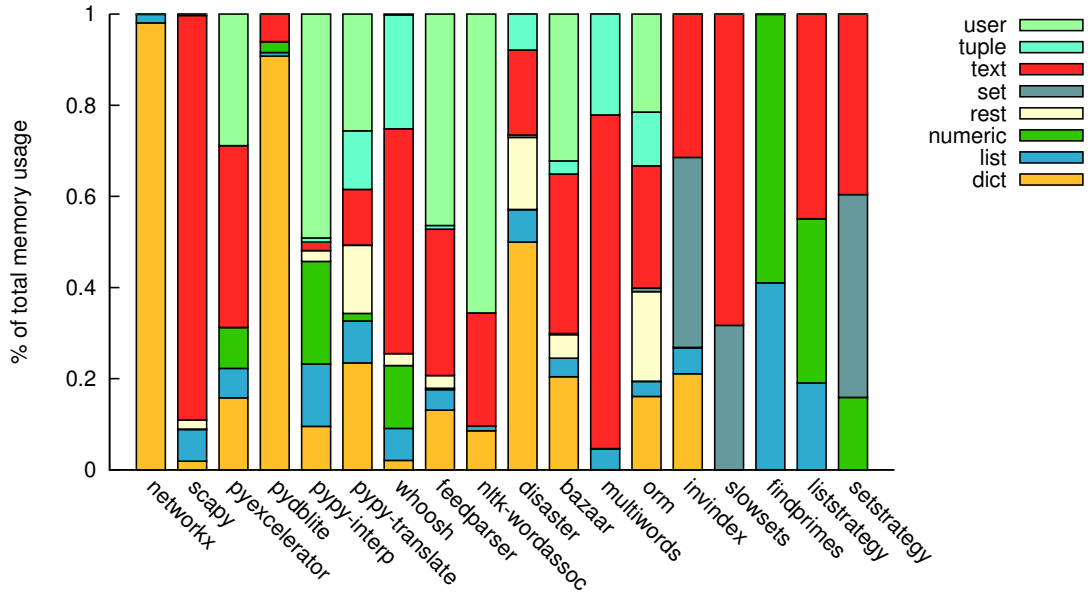

Figure 3: Percentage of memory used by different data types compared to total memory usage.

Figure 3 shows the total memory usage of each benchmark divided into the different data types. Those include lists, sets, dictionaries, numeric values (e.g. integers, longs), text strings (e.g. ascii, unicode), user generated objects and objects that are created by the interpreter (True, None, Frame, etc). Note that these results are static measurements that

were gathered from a single snapshot during execution. Furthermore, the memory size does not include the size of the elements stored in a container. It only counts the size of the object itself, although this size increases with the number of elements. It should also be noted that the category *dict* does not include the `__dict__` attribute of an object but only dictionaries created explicitly in the user program.

As we can see almost all programs make heavy use of the `dict` type. This is expected as this type is very popular in the Python language. Another type that is often used is string, mostly in the same benchmark where dictionaries are used, presumably because they are preferably used as keys. Tuples and sets seem to be of the lesser used types, whereas lists do make up a fair amount of memory compared to the total usage. In the following figures, we have a closer look on the data types that are stored in these containers. For the analysises the containers were separated into different groups. For each group it is shown how much memory is used by a container which elements only have the data type of the that group. Containers that store different data types or data types that are not interesting for this analysis (since they can not be optimized, e.g. user types) are categorized under "rest".

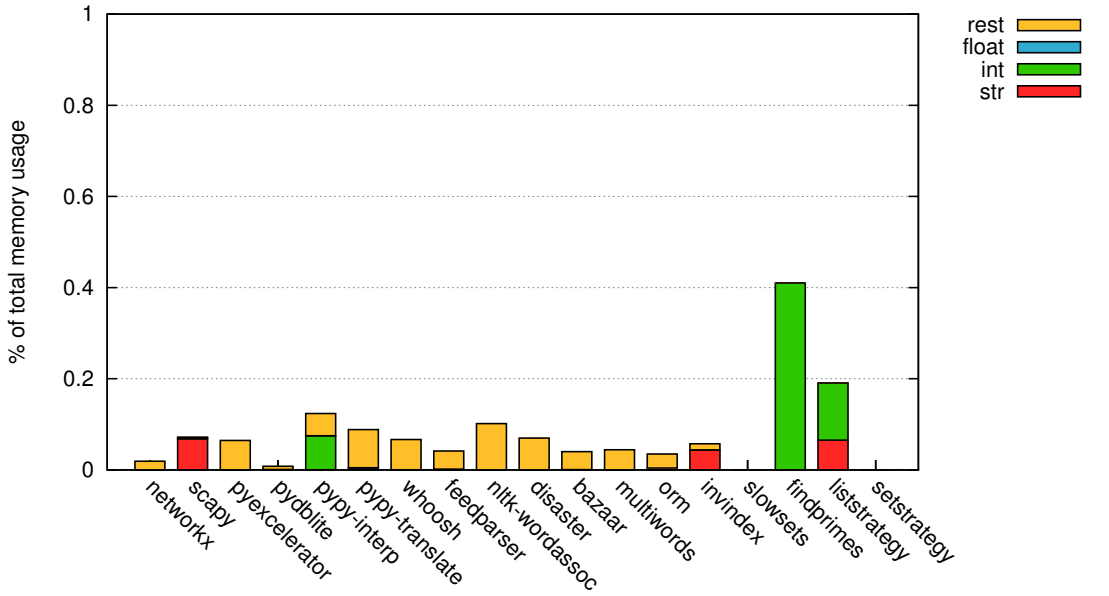

Figure 4: Data types stored in lists

Most of the benchmarks do not use as many lists as one might expect and if they do mostly arbitrary data is stored as can be seen in Figure 4. However there are still some benchmarks which have lists of strings and integers taking up to 10% of total memory for the benchmarks pypy-interp and invindex. Since the liststrategy benchmark is a synthetic benchmark it cannot be thought of as a real use case. The findprimes benchmark, although being a very small program, would be a valid implementation for finding prime numbers and could be considered a general use case.

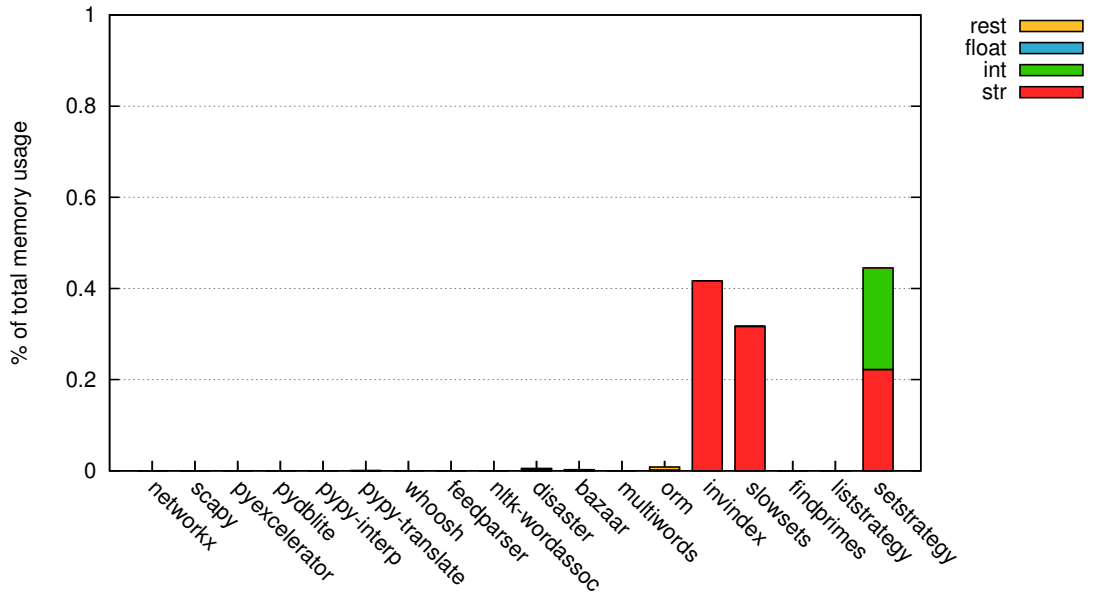

Figure 5: Data types stored in sets

The data type analysis of sets does not give many results. The reason for this could be that the benchmark selection is just an arbitrary collection of Python programs that tend to use a lot of memory. Thus, having only a few benchmarks that use sets may be pure coincidence. However, it is more likely that sets are not used as often as lists and dictionaries. Almost every benchmark uses at least some lists and dictionaries whereas sets only appear in a few (of which one was deliberately written to use sets). Nevertheless, the few benchmarks show that there are at least some use-cases where a lot of sets are used and that those sets typically contain a single data type. For instance the slowsets benchmark, which aims to find all combinations of the letters of the alphabet, uses almost 1 GB of memory.

In Figure 6 we can see that dictionaries are not only used very often but also are responsible for a good deal of the total memory usage. Furthermore, the diagram shows that the assumption that strings are often used in combination with dictionaries is mostly true. A very good example is the database software pydblite where dictionaries containing strings account for over 80% of total memory usage.

In summary, it can be said that there is some potential in saving total memory usage of a program by optimizing containers such as lists, sets and dictionaries. The memory analysis shows that these data types are used very often in Python. However, the diagrams only show the amount of containers used in the program and their percentage of total memory. They do not show how many objects a collection contains. But since the (memory) size of a container increases with the number of elements, one single container with a huge amount of elements can be as easily observed in the diagram as many containers with only few elements.

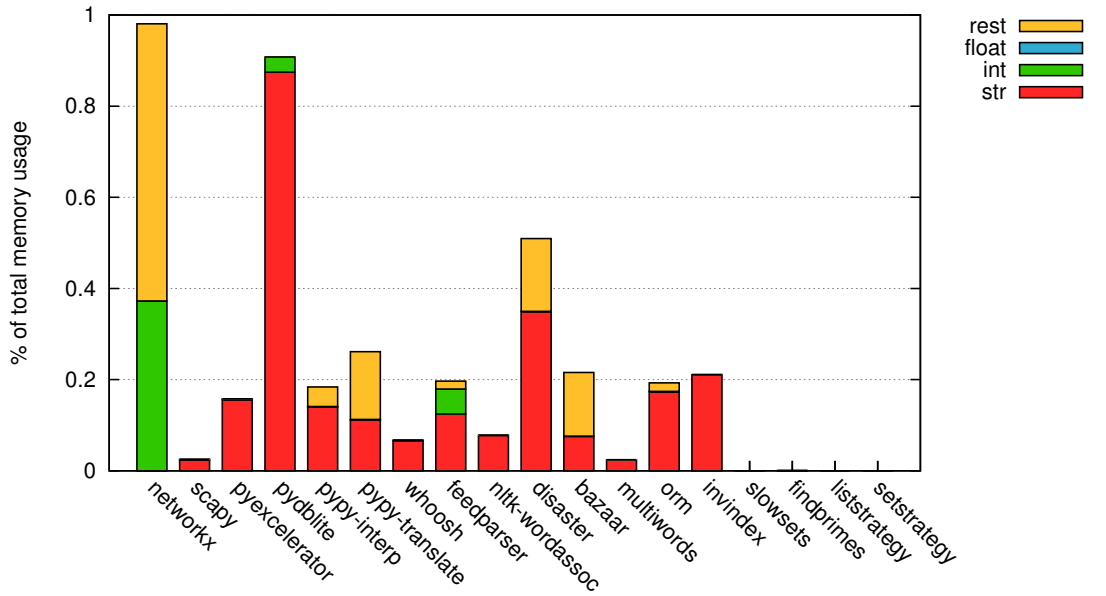

Figure 6: Data types stored in dictionaries

In Section 2.4 we have seen that data types may become quite huge due to the boxing of all values. In this section we have also seen that this is bad for collections such as lists, dicts and sets. However, another observation was that the data types in those collections are often the same, presumably because it is very unlikely to, for instance, append an integer to a huge list of strings.

4 Optimization of Data Types

Knowing that most collections do not have mixed types, those data structures could be optimized if they contain primitive data types. So instead of wrapping the elements of a list, a dictionary or a set, we want to implement these data structures in a way that they are optimized for certain primitive data types. These implementations store the content of the container in unwrapped form, getting rid of the extra indirection and boxing objects.

We start with Section 4.1 describing two possible ways to implement these optimizations on the basis of lists. In sections 4.2 and 4.3, the same idea is applied for sets and dictionaries. Since the implementation is almost the same, only differences to the list implementation and some special optimizations are described. Section 4.4 describes further optimization when interacting between different collections. Finally, section 4.5 shows the influence on the JIT.

For the further understanding of this section it is also necessary to know that in PyPy, implementations of Python objects have the prefix "`W_`", which stands for "wrapped", to distinguish them from RPython objects. For example the class representing a list is called "`W_ListObject`", the one representing a string is called "`W_StringObject`".

4.1 Lists

One approach to get lists to store their data unwrapped, would be to add a level of indirection to `W_ListObject`, making each instance point to another object that stores the actual content. For this other object, several implementations would exist: For every data type we want to store without wrapping it as well as a general one that deals with arbitrary content. The data layout would look as shown in Figure 7.

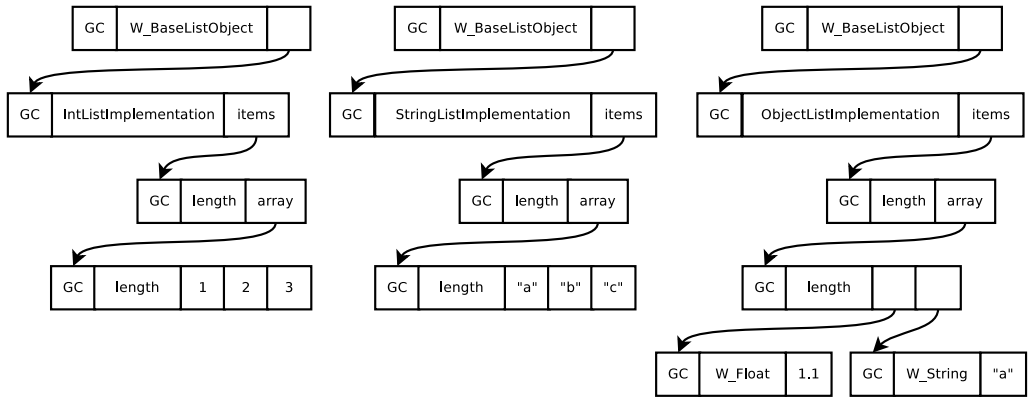

Figure 7: Class diagramm of three list objects storing some elements using special list implementations.

This approach has the problem that we need two indirections to get to the data and that the implementation instances need memory themselves.

What we would like to do is to make the `W_ListObject` point to an RPython list directly, that contains either wrapped or unwrapped data. This plan has the problem that storing different unwrapped data is not directly possible in RPython.

To solve the problem, we use the `rerased` RPython library module. It allows us to erase the type of an object, in this case a list, and returns something similar to `void *` in C, or `Object` in Java. This object is then stored on the `W_ListObject` in the field `storage`. If we want to work with the list, for instance to append or delete items, we need to unerase the storage again. An example on how to use the rerased module is shown in Listing 1.

Listing 1: Example for (un)erase

```
storage = erase([1, 2, 3, 4])
# storage is an opaque object that you can do nothing with
....
l = unerase(storage)
l.clear()
```

Now that we know how to make the `W_ListObject` point directly to wrapped or unwrapped data, we need to find out how to actually do any operations on this data. This can be accomplished by adding another field to our `W_ListObject`. This field points to a list-strategy object. The actual implementation of `W_ListObject` is now deferred to this object. For instance, a `W_ListObject` which holds only integers will use the `IntegerListStrategy`. Calling a method on a list will first invoke the method of the W_ListObject. Since the W_ListObject does not know what type its elements have, it delegates this method call to its strategy. Also it gives itself with the method as argument. This is necessary because all list-strategies are singletons and each of them can manage multiple lists. So to know on which list the operations needs to be done, it must have a reference to that list (see Listing 2 and Figure 8).

Listing 2: Delegating methods to the ListStrategies

```
class W_ListObject(W_AbstractListObject):

    def append(w_list, w_item):
        w_list.strategy.append(w_list, w_item)
```

Each strategy implements a special version of every method that can be perfomed on a list, adjusted to the type of data it manages. Examples for such methods are *append*, *pop*, *insert*, *remove*, *contains*, *index*, *getitem*, *setitem*, *getslice*, *setslice*, *extend*, *length*, *reverse*, *sort*, etc. For the most methods, like the append method, the `Integer-`, `String-` and `ObjectListStrategy` share the same implementation, since the procedure is the same. For an example on how the delegation works, let's have a look at the method `append` shown in Listing 3.

At first the strategy needs to check if the element that is appended, has the same type as the other elements in the list. Therefore, each strategy has a method called

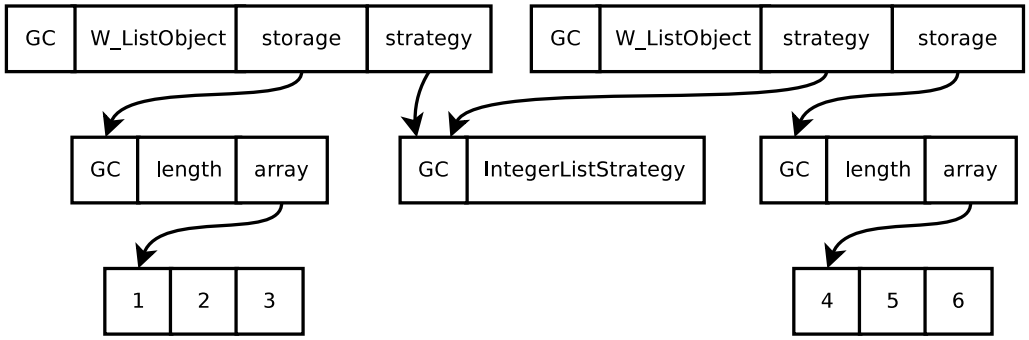

Figure 8: Using IntegerListStrategy

Listing 3: Append of the Integer- and StringListStrategy

```
class AbstractUnwrappedStrategy(object):

    def append(self,  w_list, w_item):

        if self.is_correct_type(w_item):
            self.unerase(w_list.lstorage).append(self.unwrap(w_item))
            return

        w_list.switch_to_object_strategy()
        w_list.append(w_item)
```

`is_correct_type` that returns `True` or `False` depending on the type of the given element (see Listing 4 for this method's implementation on the `IntegerListStrategy`). The `Integer-` and `StringListStrategy` have an equal implementation. The `ObjectListStrategy` always returns `True` since lists using that strategy already have different types, so every item can be added.

Listing 4: Checking the type for the IntegerListStrategy

```
class IntegerListStrategy(AbstractUnwrappedStrategy, ListStrategy):

    def is_correct_type(self, w_obj):
        return is_W_IntObject(w_obj)
```

By adding a type that is different from the elements currently stored in the list it is mandatory to change the used strategy as well as the storage in compatible ways. For example when a string is added to a list of integers the strategy will call the `switch_to_object_strategy` method which is defined on the W_ListObject (see Listing 5). There all elements contained in the list need to be wrapped and put back into the storage. Then the strategy of the list is set to the `ObjectListStrategy`.

Listing 5: Adding the wrong type changes storage and the strategy

```
class W_ListObject(W_AbstractListObject):

    def switch_to_object_strategy(self):
        if self.strategy is self.space.fromcache(EmptyListStrategy):
            list_w = []
        else:
            list_w = self.getitems()
        self.strategy = self.space.fromcache(ObjectListStrategy)

        self.init_from_list_w(list_w)
```

Currently there are only strategies for integers, strings and floats since many lists seem to store these data types. Other strategies e.g. for unicode strings are planned and can be added easily. In addition there are also two special strategies for empty lists and range-lists.

The *EmptyListStrategy* is used for storing lists without any element. For that reason the storage for a list using this strategy is `None`. The implementation of the lists methods are very simple. Since an empty list has no elements there are several operations that will not do anything. By knowing that, we can directly return the expected response without computing anything. For instance, in the former version of PyPy's list implementation, calling `contains` on an empty list would read the storage and try to iterate over it, looking for the given object. The for-loop would stop before the first iteration, noticing that there aren't any items to iterate over. But until then we already did some work. With strategies we return False at once when the method is called. For some methods it is even possible to do nothing (deleteslice, reverse), others only need to raise an `IndexError` (pop, setitem). A little more work needs to be done in case an element

is added to the empty list (see Listing 6). Since we want to use the correct strategy for the list, we need to check the type of the added element, get the corresponding strategy and initialize a new erased storage object using that strategy. Then we just assign the attributes `storage` and `strategy` of the `W_ListObject` to these values. After that, append is called again which will now be delegated to the correct strategy.

Listing 6: Append of the EmptyListStrategy

```
1 class EmptyListStrategy(ListStrategy):
2
3     def append(self, w_list, w_item):
4         self.switch_to_correct_strategy(w_list, w_item)
5         w_list.append(w_item)
```

For lists that are defined by using `range` in Python, we implement another strategy, the *RangeListStrategy*. Usually users who define range-lists need them for iteration. For that purpose it is not necessary to put all elements into the memory[2]. Instead, one should use xrange. Unfortunately, an xrange list is not a real list, thus using typical list methods on it is not possible. The `RangeListStrategy` tries to combine the advantages of both approaches. Like `xrange`, it only stores the values that define the lists, such as `start`, `stop` and `step`, while still behaving like a normal list. In contrast to CPython's where users need to specifically convert the `xrange` object to a list if they want to work with it using list methods, in PyPy users won't notice any difference at all. And as long as the user does not tamper with its content it stays like that. Only if list methods are used that destroy the initial definition of the range, such as `append` or `delete`, we need to create all elements and store them using the `IntegerListStrategy`. All of this is done internally without the user noticing that the list was altered.

Storing some of the data unwrapped by using strategies now saves one level of indirection, as can be seen in Figure 8. Of course each operation on a list needs to go via the strategy, but since we save one indirection for each element stored in that list and the strategy classes are singletons, the benefits outweigh the costs.

These optimizations have also some other advantages. Having certain data types unwrapped makes it possible to implement optimized methods for certain cases. For instance comparison of unwrapped integers is now faster than comparison between integers that are wrapped into objects. One algorithm that benefits from this optimization is the sorting of lists. Sorting needs lots of comparisons between objects. This means that for every comparison we need to determine the object's data type, unpack the wrapped object, read its value, do the same with the other object and then compare the results with each other. Now, with the data types being unwrapped, we can compare the object directly, which is much faster. To do this we need to rewrite the algorithm that is used for sorting lists and replace the general method for comparing wrapped objects with one that compares unwrapped objects directly, like integers or strings.

Another method that can be optimized is `contains` (see Listing 7). Even though

[2] with Version 3.0 of CPython the range method now works like xrange; however, lists can still be created from that intentionally using the list constructor.

its runtime is not as bad as sorting, it is still linear in the worst case. So in order to check whether an element is in the list, it must be compared to every element on the way, until a match is found. For the `Integer`- and `StringListStrategy` the method can be rewritten the same way it was done for `sort`, comparing unboxed elements if their type matches. However, it is still necessary to have a fallback method that compares wrapped elements. It is not possible to simply return False if an integer list is asked whether it contains other elements despite integers. For instance, `1.0` is a `float` and thus does not pass the type check of the `IntegerListStrategy`. However, it is still equal to `1`, so `[1, 2, 3].contains(1.0)` would return `True`. Even for strings it is not possible to predict the return value of the contain method. In dynamic languages it is very easy to define a new class that inherits from `str` and overwrites the `__eq__` method to be comparable to instances of that class (e.g. by always returning True).

Listing 7: Contains on ListStrategies

```
class AbstractUnwrappedStrategy(object):

    def contains(self, w_list, w_obj):
        if self.is_correct_type(w_obj):
            obj = self.unwrap(w_obj)
            l = self.unerase(w_list.lstorage)
            for i in l:
                if i == obj:
                    return True
        return ListStrategy.contains(self, w_list, w_obj)
```

4.2 Sets

Now we want to apply the same implementation approach we used for lists to sets. The implementation is almost the same. However, it is necessary to adjust them a little since the datastructure for sets is different. In contrast to lists where all elements are ordered and accessible by an index, the data in sets is unordered. The reason for this is that lists are usually used for iterating over objects or implementing data structures where the ordering is important (e.g. priority queues). Therefore, iterating needs to be fast but finding a certain object is not that important. Sets, in contrast, are used to store a bunch of unique objects where the ordering is not important at all but finding certain elements quickly is mandatory to implement good algorithms for comparing, intersecting or joining sets. Hence, we cannot use a (RPython) list to store the data. Instead we need an internal representation that has the mentioned properties. The data structure we are using is a dictionary. Dictionaries in Python are unordered and elements are stored using a hash map. This way elements can be accessed in constant time. Dictionaries expect `(key, value)`-pairs as elements, but since the elements of sets have only a single value, the element itself is used as the key and always maps to `None`.

Of course methods can be optimized, too, now that we have unwrapped data in our memory. In fact the potential for sets is much greater than for lists, since sets tend to do many operations in combination with other sets. Apart from the equal

method (which is similar to the one used in lists) we have methods like `difference`, `symmetric_difference`, `intersection`, `isdisjoint`, `issubset` or `issuperset`. For all these methods we can write two implementations. One for the general case where we have arbitrary objects and a specialized one for each strategy where the datatypes are unwrapped. This way we can compare integers, strings, floats, etc on a lower level which results in a great speed up compared to the former implementation.

Another advantage of having set-strategies is that we can write fast-paths for some methods. For example, let's have a look on the method `intersection`. Intersecting two sets returns a new set consisting only of elements that are contained in both of these sets. Now assume these two sets are completely different: One contains only strings. The other consists only of integers. The intersection of those sets would always be an empty set. But since usually sets may contain arbitrary objects we can not predict their content so for every object from the one set we need to check if it is also contained in the other. With set-strategies the type of the elements in a set is known, as long as the set does not use the `ObjectSetStrategy`. So, given the same scenario we could now instantly return an empty set without looking at any element at all.

To implement this behaviour we need knowledge about which strategies may have equal elements and which don't. The best way to do this is by asking the strategy itself. For instance the `IntegerSetStrategy` should return False if it is asked whether the `StringSetStrategy` may contain an equal element. On the other hand it should return True if the other strategy is the `ObjectSetStrategy`. For this purpose each strategy implements a method `may_contain_equal_element`. See Listing 8 for an example of how this method would look like for the IntegerSetStrategy.

Listing 8: may_contain_equal_elements on IntegerSetStrategy

```
1 class IntegerSetStrategy(AbstractUnwrappedSetStrategy, SetStrategy):
2
3     def may_contain_equal_elements(self, strategy):
4         if strategy is self.space.fromcache(StringSetStrategy):
5             return False
6         if strategy is self.space.fromcache(EmptySetStrategy):
7             return False
8         # add more strategies here later
9         return True
```

Now that strategies can be asked whether they and another strategy may have equal elements, the code for the intersection algorithm can be adjusted. There are now three cases which need to be taken care of differently. The first one occurs if two sets are intersected that have the same strategy. Then all elements can be compared unwrapped. The second case is the fastpath where the strategies of the two sets cannot have equal elements due to their strategy. Then it is safe to return an empty set at once. And finally there is the general case where there is no information about the other set. This is equivalent to the former implementation without strategies, meaning that all elements have to be compared wrapped. An example for the intersection method is shown in Listing 9.

Listing 9: Fastpath for W_SetObject.intersection

```
class AbstractUnwrappedSetStrategy(object):

    def _intersect_base(self, w_set, w_other):
        if self is w_other.strategy:
            strategy = self
            storage = strategy._intersect_unwrapped(w_set, w_other)
        elif not self.may_contain_equal_elements(w_other.strategy):
            strategy = self.space.fromcache(EmptySetStrategy)
            storage = strategy.get_empty_storage()
        else:
            strategy = self.space.fromcache(ObjectSetStrategy)
            storage = self._intersect_wrapped(w_set, w_other)
        return storage, strategy
```

Another method where a fast-path like this can be added is the `difference` algorithm. It returns all items that are in the one set, but not in the other. The algorithm iterates over all elements of the first set and checks for each if it is also contained in the other set. If not it is added to a new set which will be returned as the result. This algorithm has the same flaw as the former intersection method. In every case we need to iterate over all elements of the first set, even if the other set has no matching elements at all. Having two completely different sets, the difference algorithm would always return the initial set. So, as with the intersection method, we could easily add a similar fast-path to `difference`.

Two more examples where a fast-path can be added are `isdisjoint` and `issubset`. Two sets are disjoint if all elements from one set are different from the other. So for sets with different strategies we can always return True. For issubset it is the other way around. A set can only be the subset of another set, if every element from the first set is also contained in the other one. A weaker condition would be that those two sets must have at least one equal element. Since this can not be true for non-matching strategies we can return false immediately.

There are also some optimizations that do not necessarily have something to do with our set strategies. One worth mentioning is `intersection_multiple`. This method is used to intersect a set with each element of a list containing iterables such as other sets, lists or even generators. The reason for having such a method is so that users do not have to implement an intelligent way of intersecting several iterables themselves. But the former version simply iterates over the list of iterables and intersects each one with the first set. A more intelligent way would be to order the list of iterables and start with the smallest. This way we can massively reduce the amount of comparisons done during the whole algorithm. This is due to the behaviour of intersection always resulting in a set that is smaller or at least as small as the sets that were intersected. So by choosing the smallest set for the first intersection we minimize the amount of comparisons in all the following intersections. Indeed sorting the whole list of iterables is expensive, so even though we skip a lot of comparisons, this optimization might not be worth the effort. A better approach would be to only put the smallest set at the beginning. The

smallest set can be found in linear time ($\mathcal{O}(n)$) while sorting the whole list would be in $\mathcal{O}(n * log(n))$. By starting with the smallest set the amount of comparisons in all following intersections have already been heavily reduced and with every intersection the resulting set can only become smaller or stays at least the same. Sorting the list would still save some more comparisons but this would not be in proportion to the costs of the sorting.

4.3 Dictionaries

The last collection, strategies can be applied to, are dictionaries. In PyPy there already are different dictionary implementations, called `DictImplementations`. For instance there is the `StrDictImplementation`, which behaves like a normal dict but accepts only strings as key, or the `ModuleDictImplementation`, which is a cell dict implementation using a version tag. These implementations are similar to the specialized list implementations that were discussed in Section 4.1. So the first thing that needs to be done is to replace those specialized implementations by strategies and change the parent dictionary class `W_DictMultiObject` to use the correct one when initializing a new dictionary. Furthermore, it is necessary to implement two more strategies: One for the case of an empty dictionary, `EmptyDictStrategy`; and another for dictionaries with different types, `ObjectDictStrategy`.

Rewriting the specialized implementations to strategies is similar to the way the strategies for lists and sets were written. However, there are little differences. When specialized implementations were used, like the `StrDictImplementation`, it was necessary to add a way to fallback to the general implementation if another type was added that does not match. Once a dict used this general implementation there was no way to return to using other implementations. By using strategies it is now possible to switch back and forth between different implementations. For instance if a dictionary is cleared it will use the `EmptyDictStrategy` and from there will automatically pick the correct strategy depending on the first data type that is added.

Those strategies are only applied to the keys of a dictionary and not to the values. There are two reasons for this. First of all, values do change more often than keys so it is more likely for them to change their type. Second, having strategies for keys and values would result in a huge number of combinations (strings with strings/int/objects, objects with strings/ints/objects, etc) and for each a strategy would need to be implemented (or generated).

4.4 Interactions between different data types

With the optimizations for lists, sets and dictionaries, collections in PyPy became faster for local operations. However, Python allows the combination of these data structures with each other. For instance, to create a set one often uses a list to initialize it:

```
>>> set([1,2,3])
set([1,2,3])
```

Of course, this also works the other way around and there are even more combinations. Sets can be built from lists, lists can be build from sets or strings and even dictionaries can be initialized this way (except there, this is done by using the dict-method `fromkeys` which uses the elements as keys and sets the values to `None`).

The problem with this is that now the elements of these data structures are stored unwrapped for certain data types. So when a set is created from a list the elements of that list first need to be wrapped, only to become unwrapped by the set again. This happens whenever both data structures have a strategy for that particular data type.

Since all mentioned data structures are capable of dealing with unwrapped data types it would be nice to find a way to avoid the unnecessary wrapping and unwrapping when combining them with each other. All that needs to be done to achieve this is to define methods on these data structures that return their unwrapped content. Additionally in the initialization method of each data struture some functionality needs to be added to get that content and use it in the correct way.

To clearify things, here is an example: Imagine we want to be able to initialize sets with lists using unwrapped data. The first step would be to add a method to the `W_ListObject` that returns all elements unwrapped. This method needs to be added to every list strategy. For the `IntegerListStrategy` that method would look like this:

```
def getitems_int(self, w_list):
    return self.unerase(w_list.lstorage)
```

All that needs to be done is to unerase the storage and return it. The same is done for the `String`- and `FloatListStrategy`. Other strategies return `None` and range-lists need to create the real data first before returning it. The next step involves altering the `W_SetObject` to deal with the list of unwrapped data. But since a set could be initialized with any kind of iterable this would mean adding a huge amount of conditions to the init-method. First we would need to check whether the iterable is a list. Then we would need to check which strategy the list uses and then call the correct method to receive the unwrapped data. The same conditions are needed in the case that the iterable is a dictionary. To simplify this and to avoid writing a lot of conditions, a helper function can be written within the objectspace. In PyPy there already exists a method called listview which allows to convert an iterable to a list. For instance this method can be used to convert a generator into a list to initialize a set. Similar methods are now added for each data type for which strategies exist. These methods check if a given object may have unwrapped data and returns it. Otherwise it returns None. Here is an example for such a method:

```
def listview_int(self, w_obj):
    if isinstance(w_obj, W_ListObject):
        return w_obj.getitems_int()
    return None
```

In the initialization routine of the `W_SetObject` these listview methods can now be used by giving them the iterable that initalizes the set. If the method returns a list, this list can

be directly used as the new storage (for lists) or put into a new storage (for sets and dictionaries). If the listview method returns `None` the given iterable does not contain data with the specified type. So the next step would be to check for other data types using the other listview methods. If the return value of all those methods is still `None`, the iterable has arbitrary objects as data.

4.5 Influence on the JIT

Introducing strategies for collections does not only reduce memory usage but also gives the just-in-time compiler of PyPy more information to optimize generated traces. By using strategies the JIT knows the type of elements in some collections and thus can predict the type of the results when using these elements, for instance in an addition or multiplication. Or it can remove type checks that are obsolete when the type is known (e.g. when adding an integer to a list using the integer-strategy). For a more detailed example have a look at the following small Python program that stores the square of all numbers from 1 to 10000 in a list:

```
l = range(10000)
for i in l:
    l[i] = i*i
```

The optimized trace would look as follows:

```
label(TargetToken(-1221871140))
debug_merge_point('#19 FOR_ITER')
i50 = getfield_gc(p28, list.length)
i51 = uint_ge(i34, i50)
guard_false(i51, descr=<Guard17>)
p52 = getfield_gc(p28, list.items>)
i53 = getarrayitem_gc(p52, i34)
i54 = int_add(i34, 1)
debug_merge_point('#22 STORE_FAST')
debug_merge_point('#25 LOAD_FAST')
debug_merge_point('#28 LOAD_FAST')
debug_merge_point('#31 BINARY_MULTIPLY')
setfield_gc(p14, i54, W_AbstractSeqIterObject.inst_index)
i55 = int_mul_ovf(i53, i53)
guard_no_overflow(, descr=<Guard18>)
debug_merge_point('#32 LOAD_FAST')
debug_merge_point('#35 LOAD_FAST')
debug_merge_point('#38 STORE_SUBSCR')
i56 = getfield_gc(p41, list.length>)
i57 = uint_ge(i53, i56)
guard_false(i57, descr=<Guard19>)
p58 = getfield_gc(p41, list.items)
setarrayitem_gc(p58, i53, i55)
```

```
debug_merge_point('#39 JUMP_ABSOLUTE')
guard_not_invalidated(, descr=<Guard20>)
i59 = getfield_raw(160443396, pypysig_long_struct.c_value)
i60 = int_lt(i59, 0)
guard_false(i60, descr=<Guard21>)
debug_merge_point('#19 FOR_ITER')
jump(TargetToken(-122187140))
```

This trace and the trace without optimizations have some important differences. For instance the following line reads the integer from the list:

```
i62 = getarrayitem_gc(p61, i30, descr=<ArrayS 4>)
```

The optimized version returns an integer, because by using strategies the JIT knows that all objects in the list are integers. The unoptimized version, however, would return a pointer to a `W_IntObject`. This also means that before multiplying `i` with itself the JIT needs to validate that the object that was read from the list really is an integer. This is done by an additional guard:

```
guard_nonnull_class(p56, ConstClass(W_IntObject), descr=<Guard24>)
```

Then, since the JIT can not add objects with each other, it needs to read the actual value from the `W_IntObject`:

```
i58 = getfield_gc_pure(p56, W_IntObject.inst_intval)
```

Another part that is totally missing in the optimized trace is the following:

```
p70 = new_with_vtable(ConstClass(W_IntObject))
setfield_gc(p70, i66, descr=<FieldS W_IntObject.inst_intval 8>)
```

Without strategies, the result from the multiplication needs to be boxed again in a new `W_IntObject` before it can be put into the list. Having list-strategies the JIT knows that lists will store the integer unwrapped, which means that the boxing is not really needed. Therefore, the creation of the `W_IntObject` is optimized away in the first place.

5 Instances

Collections are not the only objects that store a lot of primitive data types. Primitive data is also often stored in instances. Instances in Python store two kinds of information: Their class and their attributes (also called instance variables). The amount of attributes an instance may have is not limited and in contrast to other languages such as Java and Smalltalk where all attributes are defined by the class, in Python it is even possible to add new ones. Because of this every instance has a dictionary where the names of the instance variables are mapped to their values. Since most instances of the same class will have the same set of instance variables, this way of storing them is a waste of memory. Therefore, CPython and PyPy implemented ways of fixing this problem. Whereas CPython tries to reduce memory consumption by using the idea of slots (effectively limiting the amount of variables an instance of a certain class may have), PyPy uses an approach first described in [CUL89].

5.1 Storing attributes in PyPy

To understand how the way of storing attributes in PyPy was optimized, we need to understand how they were stored before. As said before, every instance has a dictionary where all attributes are stored as key-values-pairs. Let's consider the following code:

```
class A(object):
    pass

a1 = A()
a1.x = 5

a2 = A()
a1.x = 23
```

This code creates a new class `A` and then creates two instances from it. For every instance an attribute are assigned containing an integer. The memory layout would look as shown in Figure 9.

As we can see the memory layout of instances of the same class is very similar. The only differences are the values of the integer and string objects. So in order to save memory we would like the instances to share those common parts. This is done by creating a new object, called map. A map contains the name of an attribute and an index to the value of that attribute. The value itself is stored into an array on the instance. The memory layout from before would now look as shown in Figure 10.

Each map can only store one attribute and each instance only points to one map. So adding another attribute to one of the instances is a bit more complicated. Imagine we want to add a new attribute `y` to the instance `a2`. The solution is that the `map` of an instance always points to the last attribute that was added. Currently this is the map storing the attribute `x`. Adding `y` to the instance will create a new map and move the instances pointer to that map. To be able to find the other attributes, every map also has

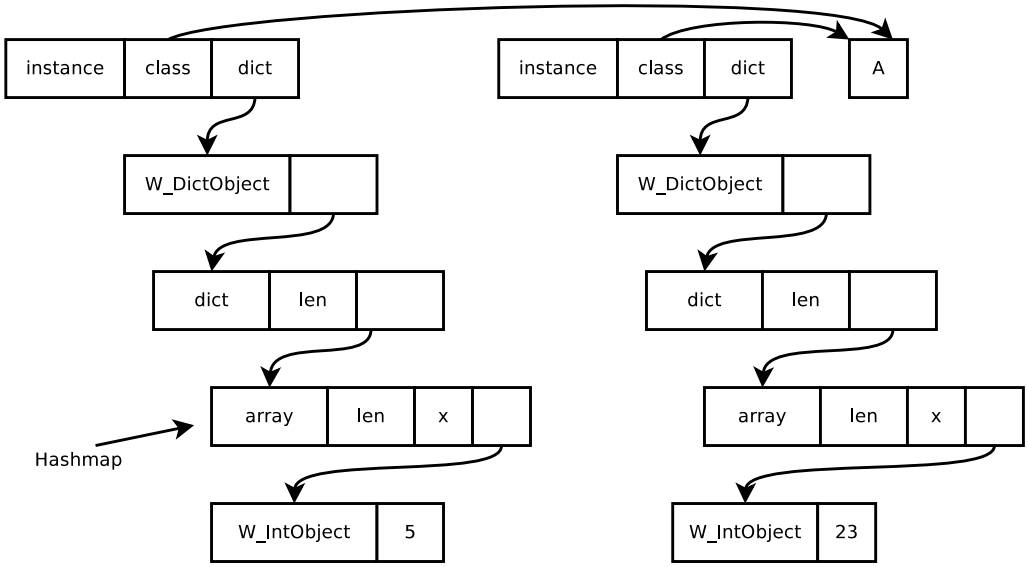

Figure 9: Class hierarchy for instances: naive approach

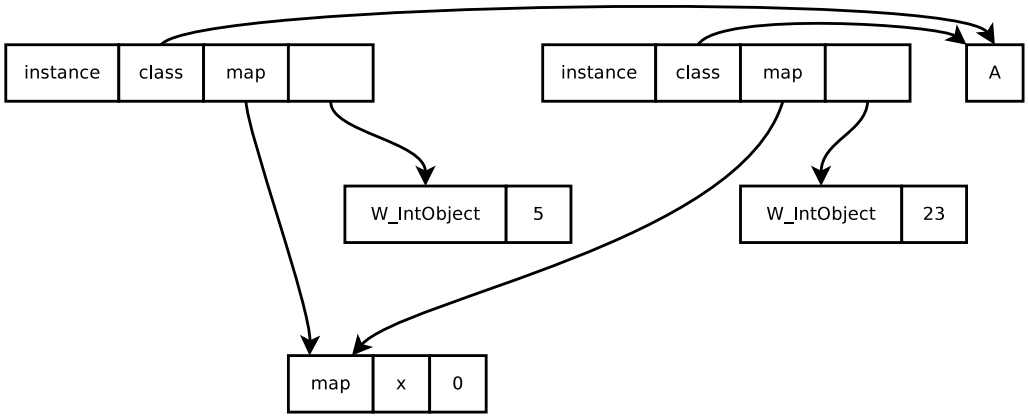

Figure 10: Class diagramm for instances: shared mapdict

a field `back` which points to another map. In this case `y` has a back pointer to `x`. Now the instance `a1` and `a2` share the map for the attribute `x` but `a2` also has another attribute `y` (see Figure 11).

As long as the order of added attributes is the same for all instances they can share all maps. So even if an attribute `y` is now added to `a1` too, the map will just be reused. However, adding another attribute, say `z`, to `a1` instead of `y` will anticipate future sharing of maps, because every attribute added afterwards cannot be reached from `a2` (see Figure 12).

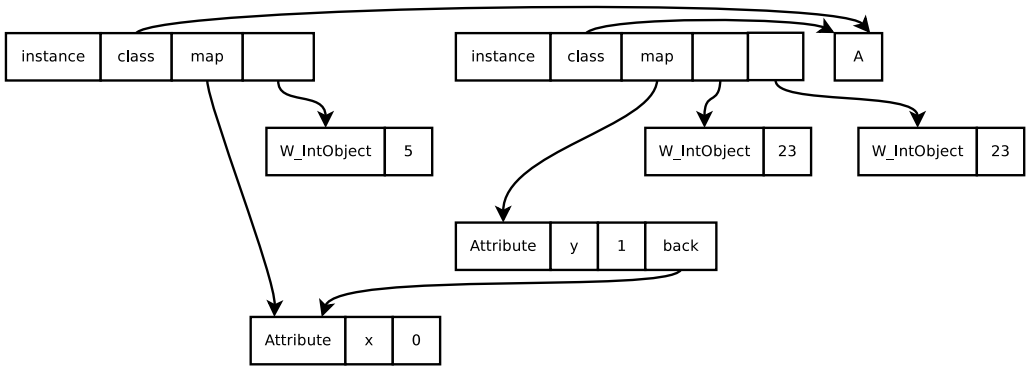

Figure 11: Shared mapdict with attributes x, y

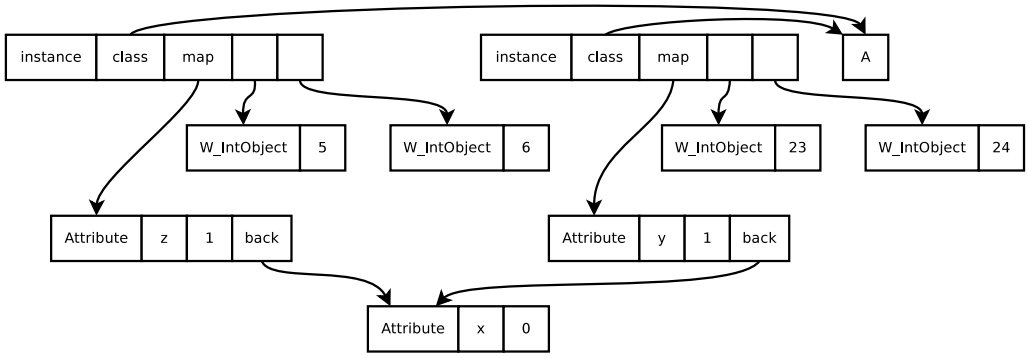

Figure 12: Shared mapdict with attributes x, y, z

5.2 Remove the wrapping

Now that we know how PyPy stores attributes on instances we can optimize those attributes even more. As we have seen, currently all attribute values are stored wrapped and are accessed through an indirection on the instance. If we want to store attribute values unwrapped, strategies as used before are not a good solution. Figure 13 shows the difference between collections and instances. Whereas in collections data of the same type is stored in a row, on instances such data is spread in a column-based manner. But as with collections, where elements often only have one type, for instances it is likely that their variables store the same type if they have the same class.

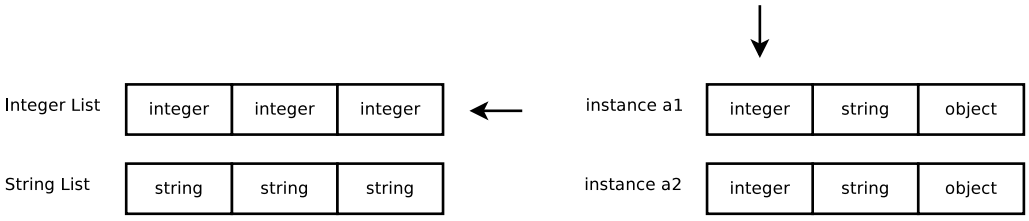

Figure 13: Row- vs column-based data

The solution to store unwrapped data is to create type specialized maps, one for each

datatype (e.g. integers, strings, floats) and a generic one for arbitrary objects. But in order to implement specialized maps it is necessary to refactor the way attributes are read and written. When we want to store data unwrapped it is mandatory to erase it, and when the data is being read it must be unerased and wrapped again. The problem is that currently the instance itself is responsible for writing and reading attributes but it doesn't and cannot know what kind of data is stored at a certain index. To solve this problem, we need to delegate the job of reading and writing the instances storage to the (specialized) maps because only they know what datatype resides behind a certain index and how to (un)erase and (un)wrap it. By using those methods, storing attributes on instances will result in a memory layout as shown in Figure 14.

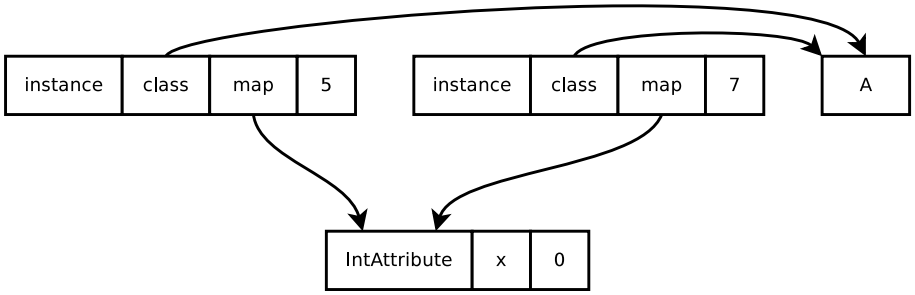

Figure 14: Unwrapped attributes

However, we need to be careful when someone wants to change the type of an attribute. With the former implementation the value of an attribute was only a pointer to an object. Therefore, switching an attribute's value from an integer to a string was not a problem. Now having unwrapped data a map can only read and write one type of data. So changing the value of an attribute is not possible without changing the map, too. This can be done by finding the old map, and replacing it by a newly created one, specialized for the type of the changed value. Since computing the new map is a complex operation it will not be described here. Figure 15 shows how the memory layout will look like after changing the attribute of one of the instances from Figure 14 to a string.

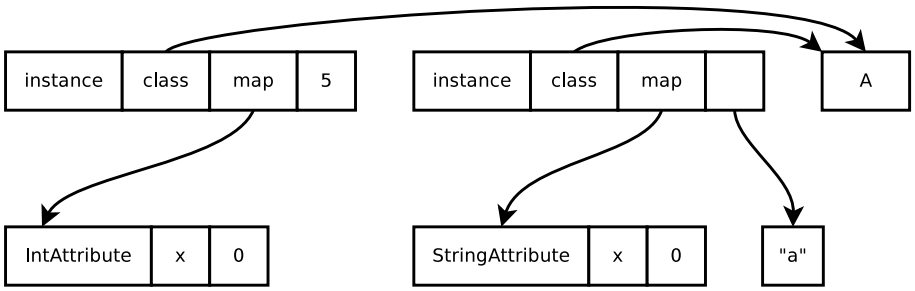

Figure 15: Changing attributes

5.3 Tagging

The optimization of the last section has one little problem. Without boxing it is not possible to store integers on the heap anymore. In dynamic languages the value of

an instance variable can be of any type and even change during execution. Thus the expected reference of a pointer is an object (e.g. in PyPy a `W_<type>Object`). But the integer cannot be stored directly on the instance either, as in statically typed languages. Of course instances could be designed in such a way that certain variables only reference a single type, but dynamic languages allow to change this type at any time. Therefore instance variables must always be pointers. However, there is a trick called tagging that solves this problem. Tagging makes it possible to store (certain) integers on an instance by replacing the pointer with the value itself. This will not only get rid of the indirection to the integer value but also saves memory because the pointer is not needed anymore.

But how does integer tagging work? On most target architectures heap blocks are aligned on multiples of 2, 4 or more adressable units to increase performance, because CPUs are optimized to handle memory that way. Therefore the least significant bits of a pointer are always zero. If we encode the integer in such a way that the least significant bit will always be 1 (for instance by using an encoding like 2n + 1) we can distinguish between a pointer and an integer stored at the same adress. The downside is that the integers we can store are limited. To be precise, the maximum integer we can store (as tagged integer) is $\frac{2^{32}}{4}$ (2^{32} is the size of 4 bytes. This is divided by 2, because we have signed integers. The other division by 2 is due to the encoding of tagged ints where we shift all bits to get a zero in the least significant bit).

Integer tagging is already implemented in PyPy. It can be used by importing `erase_int/unerase_int` from the `rlib.rerased` module. Using this in combination with type specialized maps will result in a memory layout as shown in Figure 16.

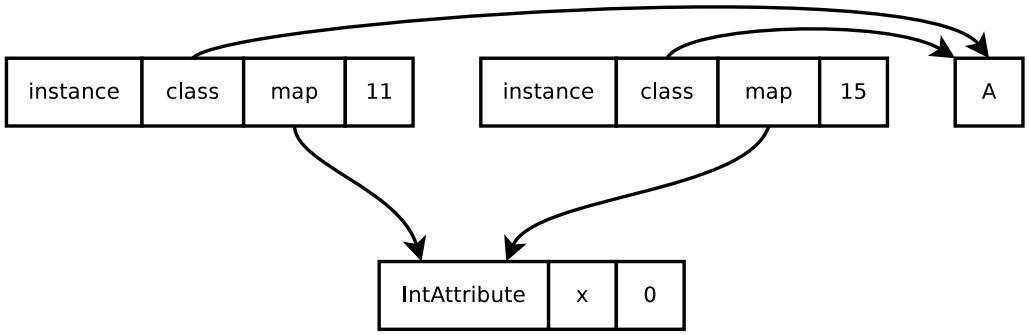

Figure 16: Maps with tagged integers

As you can see the values of the attribute `x` have changed in the memory layout. This is because of the encoding. Take `5` as an example. The binary code of that value would be `101`. To set the least significant bit to `1` this value is first shifted (`1010`) and then incremented by one (`1011`), resulting in the value `11`. The same way `7` (`111`) becomes `15` (`1111`).

6 Evaluation

To show the performance of the presented optimizations, results for several benchmarks are presented using Python and different PyPy translations. To see how each of the optimization perform separately, one PyPy version was built for each one of them, as well as one without any of the discussed optimizations and one with all optimizations activated.

All PyPy versions were built with the integrated JIT. The different versions are defined as follows:

- python: CPython, Version: 2.6.5
- none: PyPy without any of the optmizations described in this thesis. Revision: `2449ac0ea4a1`
- lists: PyPy with list strategies only. Revision: `2449ac0ea4a1`
- sets: PyPy with set strategies only. Revision: `a4bba3dd3493`
- dicts: PyPy with dictionary strategies only. Revision: `2449ac0ea4a1`
- attrs: PyPy with unwrapped attributes and integer tagging. Revision: `b15e618e3d82`
- all: PyPy with all optimizations activated (even those described in 4.4). Revision: `2449ac0ea4a1`

6.1 Hardware

The benchmarks were run on a Lenovo X60T running Ubuntu 10.04 with kernel version 2.6.32. The CPU is an x86 based Intel Dual Core 1.66Ghz processor with 2048 KB cache. RAM is 4 GB.

For CPython and PyPy with all optimizations, all benchmarks were run 30 times, taking the average of all runs so that they contain interpretation, warm-up, code-generation and execution of the generated code. The errors were calculated using a confidence interval with a 95% confidence level [GBE07]. However, since a whole run through all benchmarks with all versions and 30 iterations takes over 45 hours to complete, the benchmarks for the other PyPy versions were executed only 3 times. As can be seen, memory consumption and speed is quite stable (except for *bazaar*) so the errors will propably be the same for the other versions.

6.2 Memory

The results for the memory usage can be seen in Table 1. In Figure 17 there is also a diagram that shows the memory usage of the PyPy version with all optimizations activated normalized to PyPy as it was before those optimizations have been added.

benchmark	python	none	lists	sets	dicts	attrs	all	
networkx	114.68	117.20	125.02	127.72	57.16	115.88	54.56	± 0.36
scapy	18.81	51.34	51.18	51.14	51.10	51.12	50.69	± 0.22
pyexcelerator	168.82	195.66	189.54	195.28	174.85	186.97	175.71	± 0.06
pydblite	52.25	44.57	48.16	44.64	38.74	45.65	38.33	± 0.01
pypy-interp	86.66	97.55	93.23	96.04	90.76	97.95	88.22	± 1.86
pypy-translate	27.22	95.88	95.27	92.87	93.42	99.21	93.87	± 3.05
whoosh	15.66	52.51	51.07	51.04	50.95	51.43	50.99	± 0.32
feedparser	14.09	60.02	61.13	60.77	60.38	60.30	60.92	± 0.50
nltk-wordassoc	41.53	55.43	56.04	55.36	52.28	55.21	52.30	± 0.05
disaster	56.79	130.45	135.62	131.00	135.60	135.00	123.33	± 1.25
bazaar	63.63	176.81	181.40	182.21	186.79	190.34	176.57	± 10.93
multiwords	19.85	48.59	49.01	48.69	48.61	48.68	48.24	± 0.19
orm	32.90	75.36	77.54	75.43	75.85	78.53	74.07	± 1.46
invindex	13.03	25.52	27.45	24.64	25.71	25.50	23.75	± 0.02
slowsets	905.90	917.62	1045.64	593.88	917.64	917.69	721.56	± 0.03
findprimes	9.23	23.29	19.21	23.38	23.26	23.39	18.95	± 0.02
liststrategy	70.94	87.27	48.30	87.35	87.26	87.37	48.11	± 0.02
setstrategy	81.10	95.58	111.99	76.01	95.59	95.70	72.19	± 0.02

Table 1: Memory usage during execution in MiB

Starting with the benchmark *networkx*, we can see that the dictionary optimization saves over 50% of memory. As we have seen in Figure 3 of Section 3, over 90% of memory of the networkx benchmark was caused by dictionaries. A closer look at the container analysis reveals that most of these dictionaries are empty, which explains the good result. Whereas in the normal PyPy version each of the `W_DictObjects` creates an rpython dict, using dict-strategies the storage will be `None`. It is also notable that PyPy with the dictionary optimization performs even better than CPython, although the base memory usage of PyPy is three times bigger than in CPython. However, looking at the list optimizations shows that it is even worse than the one with list optimizations deactived. The reason for this is that with list-strategies, every list needs an extra word to point to the strategy, so having a huge amount of lists that all use the object strategy will result in more memory usage.

The next benchmark is *scapy*. It can be seen that none of the optimizations achieves any memory saving. Although the memory analysis shows that there is a lot of strings and even some lists, a closer look at the container analysis shows that actually only 1% of these strings are contained in lists.

In the *pyexcelerator* benchmark we can see that each of the list, dictionary and attribute optimization saves a little memory. However the version with all optimizations together only saves as much memory as the dictionary optimization, which saves the most of the three versions. The explanation for this is that the optimized objects are shared. If a list and a dictionary point to the same string, optimizing that string will remove the wrapper object only once. Both collections will now point to the unwrapped string, so the memory gain is the same as with only one of the optimizations. Obviously,

most of the objects are contained in the dictionaries which is why the dict optimization saves up the most.

In the *pydblite* benchmark dictionary optimizations save 15% of memory. Again a look at the memory analysis shows that this is due to the heavy usage of dictionaries. Other optimizations do not contribute to the memory saving.

The *pypy-interp* benchmark shows how well different optimizations work together. As can be seen in the memory usage table, list optimizations save approximately 4MB and dictionaries ∼5MB. Put together both optimizations save ∼9MB of memory, which is 11% of total usage.

pypy-translate is another benchmark where the optimizations do not work very well, although it uses a resonable amount of dictionaries, lists and strings. A closer look at the container analysis reveals that lists mostly store arbitrary objects and only a small amount of dicts seem to store some strings.

Yet another benchmark that does not profit by the optimizations is *whoosh*. Again the reason for this is that only a few of the many strings whoosh uses are stored inside any of the optimized collections (from almost 3 million unicode strings and 1.2 million ascii strings, only 10.000 are stored in lists or used as dictionary keys. The rest is scattered in tuples, dictionary values and other objects).

feedparser also does not save memory by using any of the optimizations. Looking at the memory analysis we can see that feedparser uses a lot of user generated objects. Indeed a closer look shows that over 45% of memory is used by user objects and unicode strings. Only 7% are dictionaries containing strings.

The *nltk-wordassoc* benchmark only has a little memory improvement by the optimizations. The reason for this can be seen if we have a look at the memory analysis. Over 60% of memory come from user generated objects. However the dict optimization still manages to save 5% of total memory.

The next benchmark is *disaster*. This benchmark is notable in the way that the optimizations only achieve an improvement when activated all at once. This effect needs further investigation.

bazaar is a benchmark where it is hard to say whether optimizations improve memory consumption or change it to the worse. Since the measurements are very unstable this benchmark needs further investigation, too.

The *multiwords* benchmark seems to have a lot of potential for optimizations using strings, when looked at the diagrams from Section 3. Unfortunately most of the strings are stored in tuples which is why the optimizations do not bring any improvement.

Another benchmark where memory savings are very low is *orm*. Only with dictionary optimizations it is possible to save 5% of total memory usage.

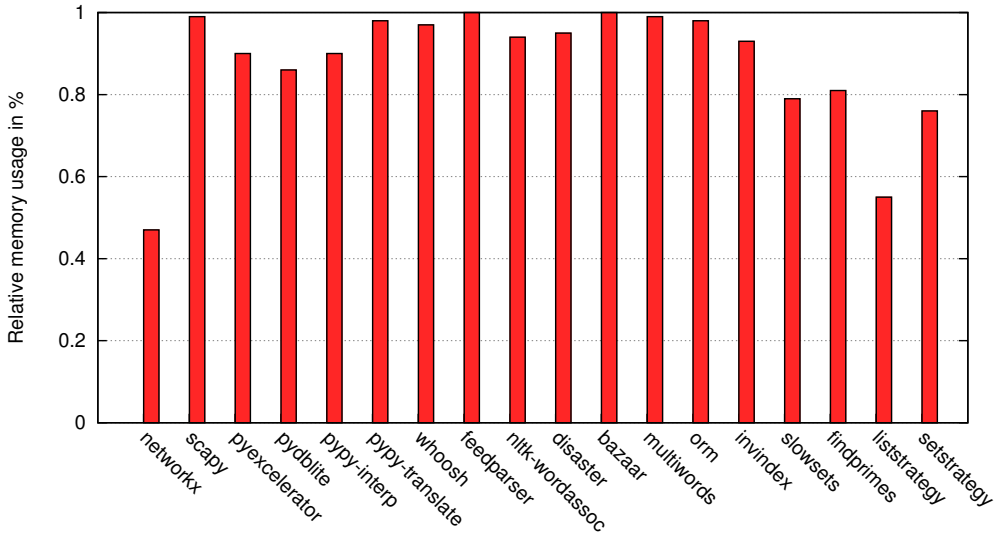

Figure 17: Relative memory usage of version *all*, normalized to version *none*

invindex is a benchmark that does only save approximately 7% of memory by using all memory optimizations. However, these optimizations will greatly increase performance as we will see later.

A benchmark that uses a lot of memory is *slowsets*. Here we can see how much the optimizations can influence memory usage, saving almost 22% of total memory compared to PyPy and even CPython.

The *findprimes* benchmark is a benchmark that also heavily uses lists. Thus, memory savings are quite high with nearly 22% of total memory.

The benchmarks *list-strategy* and *set-strategy* are microbenchmarks that were written to test the effectiveness of the list- and set-optimizations. They are not considered to be real programs and should only show the results in the best/worst case. The results show that optimizations are capable of saving up to 45% of memory with lists, and 25% with sets.

6.3 Speed

In this thesis not only memory usage was optimized but also execution time. The usage of strategies made it possible to add fastpaths to a lot of methods from lists, sets and dictionaries. Also, being able to deal with unwrapped data a lot of other things could be made faster, like comparison or assignment. Therefore, this section presents the runtime of all benchmarks compared to CPython and the former PyPy version without any of the optimizations. The results for all versions can be seen in Table 2 as well as a comparison between PyPy with all optimizations and no optimization, shown in Figure 18.

benchmark	python	none	lists	sets	dicts	attrs	all	
networkx	7.10	7.56	6.69	6.43	3.97	6.04	3.25	± 0.54
scapy	2.53	5.63	5.07	5.07	4.70	4.87	4.53	± 0.53
pyexcelerator	92.54	37.89	38.63	36.84	35.67	37.41	36.45	± 0.31
pydblite	1.47	1.18	1.22	1.17	1.07	1.18	0.98	± 0.04
pypy-interp	54.99	28.96	28.54	28.55	26.85	35.32	26.87	± 3.02
pypy-translate	11.52	30.59	29.68	29.00	29.98	31.08	30.06	± 1.19
whoosh	5.92	5.08	4.64	4.68	4.62	4.81	4.54	± 0.22
feedparser	5.87	12.08	11.92	11.81	11.83	11.86	12.19	± 1.05
nltk-wordassoc	14.32	7.23	6.60	6.20	6.13	6.38	5.80	± 1.09
disaster	108.35	65.55	68.10	65.64	64.48	66.34	65.86	± 0.46
bazaar	586.62	328.05	350.52	388.34	358.86	346.50	348.21	± 10.66
multiwords	4.69	4.68	4.63	4.71	4.67	4.69	4.54	± 0.02
orm	24.07	27.53	27.64	27.63	26.64	28.27	27.05	± 0.49
invindex	52.67	91.36	107.30	91.72	92.47	92.72	8.04	± 0.11
slowsets	93.32	148.99	159.77	175.71	147.47	187.56	113.79	± 0.77
findprimes	19.88	2.22	1.27	2.22	2.20	2.16	1.23	± 0.04
liststrategy	1.43	1.80	0.85	1.79	1.79	1.82	0.81	± 0.01
setstrategy	1.89	2.52	2.78	1.91	2.50	2.49	1.75	± 0.01

Table 2: Execution time in seconds

As we have already seen in the memory results, some benchmarks do not work well with the optimizations, resulting in very little or almost zero memory reduction. In most of the benchmarks this also means that there is also no speedup. These benchmarks are *scapy, pyexcelerator, pypy-translate, whoosh, feedparser, bazaar, multiwords* and *orm*.

Benchmarks where we can find small, but still good improvements of execution time are *scapy* (12% faster), *pydblite* (14%), *pypy-interp* (7%), *nltk-wordassoc* (8%) and *disaster* (5%).

Some benchmarks perform much better. Of course there are the microbenchmarks *list-strategy* (55%) and *set-strategy* (29%). But also some real programs like *networkx*, which is almost 60% faster. We can see in Table 2 that this can mostly be attributed to the dictionary optimizations and the optimizations described in Section 4.4. Another one is *slowsets* where set optimizations are responsible for 24% speed up. *findprimes* is also a good example that shows that list-strategies can reduce execution time by over 40%. However, the best results can be seen in the *invindex* benchmark. Although memory usage is not reduced much by the optimizations, execution time is 90% faster. The reason for this is that the algorithm that computes the inverted index frequently creates sets from lists, which was optimized in Section 4.4.

Since all these benchmarks were specifically choosen for this thesis, it is reasonable to have a look at some other benchmarks, too. Table 3 and Figure 19 show the results of PyPy's speed benchmarks. As with the results from above, it can be seen that for many programs the optimizations presented in this thesis do not achieve better results, but for some benchmarks they work quite well.

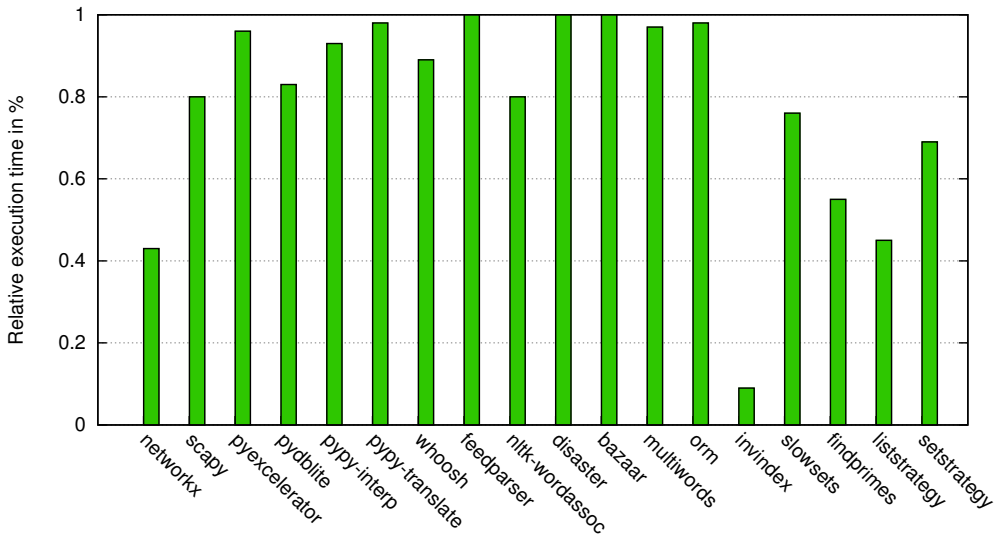

Figure 18: Execution time of version *all*, normalized to version *none*

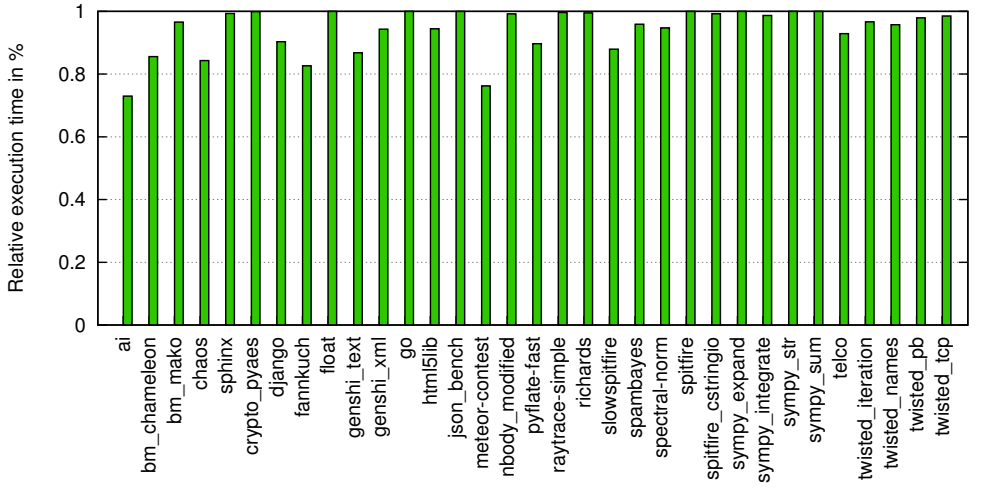

Figure 19: Execution time of version *all*, normalized to version *none* (PyPy benchmarks)

benchmark	none		all	
ai	0.2413	± 0.0968	0.1761	± 0.0974
bm_chameleon	0.0578	± 0.0525	0.0495	± 0.0450
bm_mako	0.1479	± 0.0447	0.1428	± 0.0432
chaos	0.0439	± 0.1276	0.0370	± 0.1276
sphinx	384.7070		382.0640	
crypto_pyaes	0.2004	± 0.1004	0.2002	± 0.1012
django	0.1277	± 0.0175	0.1154	± 0.0165
fannkuch	0.5973	± 0.0175	0.4937	± 0.0178
float	0.1398	± 0.0454	0.1409	± 0.0463
genshi_text	0.0827	± 0.1200	0.0718	± 0.1203
genshi_xml	0.1903	± 0.2527	0.1795	± 0.2546
go	0.2755	± 0.1691	0.2858	± 0.2174
html5lib	10.2491	± 5.7234	9.6805	± 5.1664
json_bench	5.3425	± 0.2416	5.3540	± 0.2600
meteor-contest	0.4472	± 0.0294	0.3409	± 0.0332
nbody_modified	0.1227	± 0.0144	0.1217	± 0.0142
pyflate-fast	1.8605	± 0.0548	1.6684	± 0.0584
raytrace-simple	0.0852	± 0.0102	0.0849	± 0.0136
richards	0.0132	± 0.0103	0.0132	± 0.0104
slowspitfire	1.1285	± 0.0237	0.9924	± 0.0253
spambayes	0.2628	± 0.2217	0.2520	± 0.2184
spectral-norm	0.0450	± 0.0266	0.0426	± 0.0268
spitfire	6.8732	± 0.3339	7.1548	± 0.3226
spitfire_cstringio	5.3866	± 0.2172	5.3452	± 0.1878
sympy_expand	2.1035	± 2.0126	2.1314	± 2.0531
sympy_integrate	5.7656	± 5.6782	5.6902	± 5.7986
sympy_str	1.7621	± 2.2802	1.7686	± 2.3327
sympy_sum	1.4028	± 0.9527	1.4277	± 0.9619
telco	0.1331	± 0.0886	0.1237	± 0.0899
twisted_iteration	0.0339	± 0.0001	0.0328	± 0.0001
twisted_names	0.0054	± 0.0002	0.0052	± 0.0003
twisted_pb	0.0462	± 0.0024	0.0453	± 0.0022
twisted_tcp	1.2741	± 0.0479	1.2553	± 0.0554

Table 3: Execution time of PyPy benchmarks in seconds

7 Related Work

The optimization for instance variables in PyPy described in Section 5 was intially explored by Chambers, Ungar and Lee [CUL89]. For their implementation of an efficient VM for SELF, they introduced maps to share attributes between prototypes of the same clone family.

There has been also a lot of other related work in the context of Java Virtual Machines. Most of this work also tries to target memory constrained mobile devices and is not targeted at dynamic languages specifically. Therefore it does not exploit the specific memory effects of dynamic languages.

In [SHM08], Sata, Hirzel and McKinley explore "the sources and types of memory inefficiencies in a set of Java benchmarks", outlining "previously proposed memory-saving approaches and idealized heap compaction". All these optimizations could be easily applied to PyPy. Although not all of them would achieve enough memory reduction to justify the implementation effort.

They start with compression techniques that operate on objects. One technique is the *strictly-equal object sharing* proposed by Appel and Gonçalves in [AG93]: If two objects are stricly equal, i.e. if they have the same class and all fields have the same value, then they can completely share their memory. For instance, by only allocating one instance and point all references to that instance. The same technique can also be used for arrays where all elements are the same. Another technique is *deep-equal object and array sharing* which works almost the same as *strictly-equal object sharing*, though in an eased form. To share memory, two objects (or arrays) do not necessarily must have exactly the same pointer fields (or elements) as long as the targets of the pointers (the elements) are equal. Therefore, this compression technique is even more efficient than the stricly-equal one.

Other compression techniques work on fields of class instances. Asanian and Rinard explored *constant field elision*, *field bit-width reduction* and *dominant-value field hashing* [AR03]. Constant field elision reduces memory by making fields static that have the same value for all instances. This way there is only one value on the heap that is referenced by all those instances. Field bit-width reduction represents fields with smaller bit-width if their values are small (e.g. storing more than one value in the allocated memory of one field). Dominant-value field hashing takes advantage of instances which field values have only few distinct values. Those values can then be stored into a dictionary and referenced by their index that is stored on the field instead. Aberrant values are stored in a hashmap with the object ID as key.
Chen, Kandemir and Irwin exploit frequently occuring field values in instance representations to store instances in a more compact way [CKI05]. Their implementation is based on the same idea as dominant-value field hashing, instead that they use offline profiling and per-instance decisions to deal with mistakes. Also, instead of optimizing single fields of instances, they look at all fields of a class together [SHM08].
If fields of instances do not have all the same value but the amount of those values is small ($K < 256$), *field value set indirection* can be used [CR07, TABP07]. Then instead of the value, the index into a dictionary is stored on the fields and the actual values are stored in the dictionary. If the amount of values is bigger ($K > 256$), the 255 most frequent values are stored in the dictionary and the rest is stored in a hashmap, index by the object ID.

The *lazy invariant computation* optimizes fields in a way that for two fields that are always identical only one value is stored. Furthermore, if a field is the result of the computation of two other fields, then this result is not stored at all, since it can always be computed when needed.

A compression technique that operates on array instances is *trailing zero array trimming*. This technique works against the "over-provision of the capacity of arrays used as buffers that" that lead "to unused trailing zeros" by trimming them [CKV+03]. This technique is less useful in dynamic languages, because most dynamic languages provide resizable array lists which users do not need to over-allocate themselves. Another one is *array bit-width reduction* that works like *field bit-width reduction*. It works, for example, on boolean arrays. Instead of storing booleans into an array it uses a bit vector for representation. Another use case proposed by Zilles are character arrays that are represented by a 16-bit encoding (unicode). If in such an array all characters only need 8-bit the character array can be replaced by a byte array [Zil07]. Two other techniques are *array value set indirection* and *array value set caching* which work similar to the set indirection and caching of fields from above.

8 Conclusion and Future Work

This thesis discussed some advantages and disadvantages of Dynamic Languages. One major problem was that they use a lot of memory. To see where this memory comes from and how it is partitioned, a selection of programs with heavy memory usage where analyzed. The results showed that a lot of data comes from primitive data types such as strings, integers and floats and that these are often stored in collections like lists, sets and dictionaries. It was explained that data in dynamic languages is typically boxed and why this is necessary. But it was also explained that one can reduce some overhead by removing that boxing, which is possible if only one type of data is stored in a collection. Section 4 presented a feasible implementation for this and some coherent speed optimizations on top of PyPy. Another optimization was presented in Section 5, where it was shown how it is possible to store unwrapped data on instance variables. To measure how well the optimizations perform, benchmarks were written that analyzed memory usage and execution time of the same programs that were also used for the memory analysis. The results showed that there are many cases were the presented optimizations can reduce memory usage and execution time between 10% to 90%. However, there were also some programs where no improvement was achieved.

This thesis shows that there are many possibilites to optimize dynamic languages and the results show that they have great potential to reduce memory usage as well as execution time. There are still a lot of optimizations that haven't been implemented yet. For instance, the optimizations for collections can be extended to support more data types like floats, which were only added for lists. Furthermore, there are still some fastpaths that could be written, for instance to make comparison (e.g. eq, lt, gt) faster for lists. Then there is a possible optimization for strings which could check whether a unicode string really contains only ascii characters and then stores it as an ascii string. Another collection that could be optimized are tuples which can be specialized depending on their length and even on often used combinations of types.

Acknowledgement

I would like to thank Carl Friedrich Bolz for his constant support during the implementation and writing of this thesis. Also, I would like to thank Christian Boland, Sven Hager and David Schneider for many inspiring discussions and Armin Rigo for his help with PyPy related topics.

References

[AACM07] ANCONA, Davide ; ANCONA, Massimo ; CUNI, Antonio ; MATSAKIS, Nicholas D.: RPython: A Step towards Reconciling Dynamically and Statically Typed OO Languages. (2007), 53-64. http://dx.doi.org/10.1145/1297081.1297091. – DOI 10.1145/1297081.1297091. ISBN 978–1–59593–868–8

[AG93] APPEL, Andrew W. ; GONÇALVES, Marcelo J.: *Hash-consing Garbage Collection*. 1993

[AR03] ANANIAN, C. S. ; RINARD, Martin: Data size optimizations for java programs. In: *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. ACM (LCTES '03). – ISBN 1–58113–647–1, 59–68

[Bay12] BAYER, Mike: *Orm2010*. http://techspot.zzzeek.org. Version: 2012. – [Online; accessed 13-February-2012]

[BDB00] BALA, Vasanth ; DUESTERWALD, Evelyn ; BANERJIA, Sanjeev: Dynamo: A Transparent Dynamic Optimization System. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (2000), Aug. http://portal.acm.org/citation.cfm?id=349299.349303

[CKI05] CHEN, Guangyu ; KANDEMIR, Mahmut ; IRWIN, Mary J.: Exploiting frequent field values in java objects for reducing heap memory requirements. In: *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM (VEE '05). – ISBN 1–59593–047–7, 68–78

[CKV+03] CHEN, G. ; KANDEMIR, M. ; VIJAYKRISHNAN, N. ; IRWIN, M. J. ; MATHISKE, B. ; WOLCZKO, M.: Heap compression for memory-constrained Java environments. In: *SIGPLAN Not.* 38 (2003), October, 282–301. http://dx.doi.org/http://doi.acm.org/10.1145/949343.949330. – DOI http://doi.acm.org/10.1145/949343.949330. – ISSN 0362–1340

[Cod12] CODE, Rosetta: *Inverted Index*. http://rosettacode.org/wiki/Inverted_index#Python. Version: 2012. – [Online; accessed 14-February-2012]

[CR07] COOPRIDER, Nathan D. ; REGEHR, John D.: Offline compression for on-chip ram. In: *SIGPLAN Not.* 42 (2007), June, 363–372. http://dx.doi.org/http://doi.acm.org/10.1145/1273442.1250776. – DOI http://doi.acm.org/10.1145/1273442.1250776. – ISSN 0362–1340

[CUL89] CHAMBERS, C. ; UNGAR, D. ; LEE, E.: An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In: *SIGPLAN Not.* 24 (1989), September, 49–70. http://dx.doi.org/http://doi.acm.org/10.1145/74878.74884. – DOI http://doi.acm.org/10.1145/74878.74884. – ISSN 0362–1340

[Dev12a] DEVELOPERS, NetworkX: *NetworkX*. http://networkx.lanl.gov. Version: 2012. – [Online; accessed 06-February-2012]

[Dev12b] DEVELOPERS, Whoosh: *Whoosh*. https://bitbucket.org/mchaput/whoosh. Version: 2012. – [Online; accessed 06-February-2012]

[Dev12c] DEVELOPERS pyExcelerator: *pyExcelerator*. http://sourceforge.net/projects/pyexcelerator. Version: 2012. – [Online; accessed 03-February-2012]

[Doc12] DOCUMENTATION, PyPy: *Object Spaces*. http://doc.pypy.org/en/latest/objspace.html. Version: 2012. – [Online; accessed 13-February-2012]

[FM08] FLANAGAN, David ; MATSUMOTO, Yukihiro: *The ruby programming language*. First. O'Reilly, 2008. – ISBN 9780596516178

[GBE07] GEORGES, Andy ; BUYTAERT, Dries ; EECKHOUT, Lieven: Statistically Rigorous Java Performance Evaluation. In: *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (2007), Oct. http://portal.acm.org/ft_gateway.cfm?id=1297033&type=pdf&coll=DL&dl=GUIDE&CFID=12089051&CFTOKEN=21716971

[Goe00a] GOETHE, Johann W.: *Faust: Der Tragödie erster Teil*. Project Gutenberg, 2000

[Goe00b] GOETHE, Johann W.: *Faust: Der Tragödie zweiter Teil*. Project Gutenberg, 2000

[Gro12] GROUP, The P.: *PHP*. http://www.php.net. Version: 2012. – [Online; accessed 13-February-2012]

[Gud93] GUDEMAN, David: *Representing Type Information in Dynamically Typed Languages*. 1993

[Int99] INTERNATIONAL, ECMA: *Standard ECMA-262* http://www.ecma-international.org/publications/standards/Ecma-262.htm

[Log12] LOGILAB: *Pylint*. http://www.logilab.org/project/pylint. Version: 2012. – [Online; accessed 03-February-2012]

[Lut96] LUTZ, Mark: *Programming Python*. First. O'Reilly, 1996. – ISBN 1–56592–197–6

[Mad07] MADNANI, Nitin: Getting started on natural language processing with Python. In: *Crossroads* 13 (2007), September, Nr. 4, 5. http://dx.doi.org/10.1145/1315325.1315330. – DOI 10.1145/1315325.1315330. – ISSN 1528–4972

[Nlt12] *Natural Language Toolkit*. http://www.nltk.org. Version: 2012. – [Online; accessed 13-February-2012]

[Que12] QUENTEL, Pierre: *PyDbLite*. http://www.pydblite.net. Version: 2012. – [Online; accessed 03-February-2012]

[RP06] RIGO, Armin ; PEDRONI, Samuele: PyPy's approach to virtual machine construction. In: *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (2006), Oct. http://portal.acm.org/citation.cfm?id=1176617.1176753

[RP10] RADZISZEWSKI, Adam ; PIASECKI, Maciej: A preliminary Noun Phrase Chunker for Polish. In: *Intelligent Information Systems*, Springer, 2010, S. 169–180. – Chunker available at http://nlp.pwr.wroc.pl/trac/private/disaster/

[SHM08] SARTOR, Jennifer B. ; HIRZEL, Martin ; MCKINLEY, Kathryn S.: No bit left behind: the limits of heap data compression. In: *Proceedings of the 7th international symposium on Memory management*. ACM (ISMM '08). – ISBN 978–1–60558–134–7, 111–120

[SL99] SILVA, Joaquim F. ; LOPES, Gabriel P.: A local maxima method and a fair dispersion normalization for extracting multi-word units from corpora. In: *Sixth Meeting on Mathematics of Language*

[TABP07] TITZER, Ben L. ; AUERBACH, Joshua ; BACON, David F. ; PALSBERG, Jens: The ExoVM system for automatic VM and application reduction. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM (PLDI '07). – ISBN 978–1–59593–633–2, 352–362

[Tea12] TEAM, TCL C.: *TCL*. http://www.tcl.tk. Version: 2012. – [Online; accessed 13-February-2012]

[VR+94] VAN ROSSUM, G. et al.: *Python programming language*. 1994

[WCO00] WALL, Larry ; CHRISTIANSEN, Tom ; ORWANT, Jon: *Programming Perl*. 3. ed. Beijing : O'Reilly, 2000. – ISBN 0–596–00027–8

[Zil07] ZILLES, Craig: Accordion arrays: Selective compression of unicode arrays in Java. In: *In International Symposium on Memory Management*, 2007

List of Figures

List of Tables

Listings